Test-Driven Development

Teste e Design no Mundo Real

MAURICIO ANICHE

Agradecimentos

Essa talvez seja a seção mais difícil de se escrever, pois a quantidade de pessoas que participaram direta ou indiretamente do livro é muito grande.

Vou começar agradecimento meu pai, mãe e irmão, que a todo momento me apoiaram na decisão de fazer um mestrado, entender como ciência deve ser feita, e que sofreram junto comigo nos momentos de grande stress (que todo mestrado proporciona!).

Agradeço também ao meu orientador de mestrado e doutorado, prof. Dr. Marco Aurelio Gerosa, que me ensinou como as coisas funcionam "do lado de lá". Sem ele, acho que esse livro seria muito diferente; seria mais apaixonado, porém menos verdadeiro. Se meu texto olha TDD de maneira fria e imparcial, a culpa é dele.

Os srs. Paulo Silveira e Adriano Almeida também merecem uma lembrança. Mesmo na época em que a Casa do Código não existia de fato, eles já haviam aceitado a ideia do livro de TDD. Obrigado pela confiança.

Todas as pessoas das últimas empresas em que atuei também me ajudaram muito com as incontáveis conversas de corredor sobre o assunto. Isso com certeza enriqueceu muito o texto.

Agradeço também aos amigos José Donizetti, Guilherme Moreira e Rafael Ferreira, que gastaram tempo lendo o livro e me dando sugestões de como melhorar.

Por fim, obrigado a você que está lendo esse livro. Espero que ele ajude.

Quem sou eu?

Meu nome é Mauricio Aniche, e trabalho com desenvolvimento de software há por volta de 10 anos. Em boa parte desses 10 anos, atuei como consultor para diferentes empresas do mercado brasileiro e internacional. Com certeza, as linguagens mais utilizadas por mim ao longo da minha carreira foram Java, C# e C.

Como sempre pulei de projeto em projeto (e, por consequência, de tecnologia em tecnologia), nunca fui a fundo em nenhum delas. Pelo contrário, sempre foquei em entender princípios que pudessem ser levados de uma para outra, para que no fim, o código saísse com qualidade, independente da tecnologia.

Em meu último ano da graduação, 2007, comecei a ler mais sobre a ideia de testes automatizados e TDD. Achei muito interessante e útil a ideia de se escrever um programa para testar seu programa, e decidi praticar TDD, por conta própria, para entender melhor como ela funcionava.

Gostei muito do que vi. De 2007 em diante, resolvi praticar, pesquisar e divulgar melhor minhas ideias sobre o assunto. Comecei devagar, apenas blogando o que estava na minha cabeça e que gostaria de *feedback* de outros desenvolvedores. Mas para fazer isso de maneira mais decente, resolvi ingressar no programa de Mestrado da Universidade de São Paulo. Lá, pesquisei sobre os efeitos da prática de TDD no design de classes.

Ao longo desse tempo participei da grande maioria dos eventos relacionados ao assunto. Palestrei nos principais eventos de métodos ágeis do país (como Agile Brazil, Encontro Ágil), de desenvolvimento de software (QCON SP e DNAD), entre outros menores. Cheguei a participar de eventos internacionais também; fui o único palestrante brasileiro no Primeiro Workshop Internacional sobre TDD, em 2010, na cidade de Paris. Isso mostra também que tenho participado dos eventos acadêmicos. Em 2011, apresentei um estudo sobre TDD no WBMA (Workshop Brasileiro de Métodos Ágeis), e em 2012, no maior simpósio brasileiro sobre engenharia de software, o SBES.

Atualmente trabalho pela Caelum, como consultor e instrutor. Também sou aluno de doutorado pela Universidade de São Paulo, onde continuo a pesquisar sobre a relação dos testes de unidade e qualidade do código.

Portanto, esse é meu relacionamento com TDD. Nos últimos anos tenho olhado ele de todos os pontos de vista possíveis: de praticante, de acadêmico, de pesquisador, de apaixonado, de frio. Esse livro é o relato de tudo que aprendi nesses últimos anos.

Prefácio

TDD é uma das práticas de desenvolvimento de software sugeridas por diversas metodologias ágeis, como XP. A ideia é fazer com que o desenvolvedor escreva testes automatizados de maneira constante ao longo do desenvolvimento. Mas, diferentemente do que estamos acostumados, TDD sugere que o desenvolvedor escreva o teste antes mesmo da implementação.

Essa simples inversão no ciclo traz diversos benefícios para o projeto. Baterias de testes tendem a ser maiores, cobrindo mais casos, e garantindo uma maior qualidade externa. Além disso, escrever testes de unidade forçará o desenvolvedor a escrever um código de maior qualidade pois, como veremos ao longo do livro, para escrever bons testes de unidade, o desenvolvedor é obrigado a fazer bom uso de orientação a objetos.

A prática nos ajuda a escrever um software melhor, com mais qualidade, e um código melhor, mais fácil de ser mantido e evoluído. Esses dois pontos são importantíssimos em qualquer software, e TDD nos ajuda a alcançá-los. Toda prática que ajuda a aumentar a qualidade do software produzido deve ser estudada.

Neste livro, tentei colocar toda a experiência e tudo que aprendi ao longo desses últimos anos praticando e pesquisando sobre o assunto. Mostrei também o outro lado da prática, seus efeitos no design de classes, que é muito falada mas pouco discutida e explicada. A prática de TDD, quando bem usada, pode ser bastante produtiva. Mas, como verá ao longo do livro, os praticantes devem estar sempre alertas as dicas que o teste dará sobre nosso código. Aqui, passaremos por eles e o leitor ao final do livro terá em mãos uma nova e excelente ferramenta de desenvolvimento.

A quem se destina esse livro?

Esse livro é destinado a desenvolvedores que querem aprender a escrever testes de maneira eficiente, e que desejam aprender a como melhorar ainda mais o código

que produzem. Neste livro, utilizamos Java para demonstrar os conceitos discutidos, mas você pode facilmente levar as discussões feitas aqui para a sua linguagem de programação favorita. Mesmo que você já pratique TDD, tenho certeza que aqui encontrará discussões interessantes sobre como a prática dá *feedback* sobre problemas de acoplamento e coesão, bem como técnicas para escrever testes melhores e mais fáceis de serem mantidos.

Testadores também podem se beneficiar deste livro, entendendo como escrever códigos de teste de qualidade, quando ou não usar TDD, e como reportar problemas de código para os desenvolvedores.

Como devo estudar?

Ao longo do livro, trabalhamos em diversos exemplos, muito similares ao mundo real. Todo capítulo possui sua parte prática e parte teórica. Na parte prática, muito código de teste é escrito. Na parte teórica, refletimos sobre o código que produzimos até aquele momento, o que foi feito de bom, o que foi feito de ruim, e melhoramos de acordo.

O leitor pode refazer todos os códigos produzidos nos capítulos. Praticar TDD é essencial para que as ideias fiquem naturais. Além disso, a Caelum também disponibiliza um curso online sobre testes automatizados [9], que pode ser usado como complemento desse livro.

Boa leitura!

Sumário

Versão: 16.2.5

Capítulo 1

Introdução

Será que testar software é realmente importante? Neste capítulo, discutimos um pouco sobre as consequências de software não testado e uma possível solução para isso, que hoje é um problema para a sociedade como um todo, já que softwares estão em todos os lugares.

1.1 Era uma vez um projeto sem testes...

Durante os anos de 2005 e 2006, trabalhei em um projeto cujo objetivo era automatizar todo o processo de postos de gasolina. O sistema deveria tomar conta de todo o fluxo: desde a comunicação com as bombas de gasolina, liberando ou não o abastecimento, até relatórios de fluxo de caixa e quantidade de combustível vendido por dia ou por bomba.

A aplicação era desenvolvida inteira em C e deveria rodar em um microdispositivo de 200Mhz de processamento e 2MB de RAM. Nós éramos em 4 desenvolvedores, onde todos programavam e todos testavam ao mesmo tempo. Os testes não eram tão simples de serem feitos, afinal precisávamos simular bombas de

gasolinas, abastecimentos simultâneos, e etc. Mas, mesmo assim, nos revezávamos e testávamos da forma que conseguíamos.

Após alguns meses de desenvolvimento, encontramos o primeiro posto disposto a participar do piloto da aplicação. Esse posto ficava em Santo Domingo, capital da República Dominicana.

Eu, na época líder técnico dessa equipe, viajei para o lugar com o objetivo de acompanhar nosso produto rodando pela primeira vez. Fizemos a instalação do produto pela manhã e acompanhamos nosso "bebê" rodando por todo o dia. Funcionou perfeitamente. Saímos de lá e fomos jantar em um dos melhores lugares da cidade para comemorar.

Missão cumprida.

1.2 Por que devemos testar?

Voltando do jantar, fui direto pra cama, afinal no dia seguinte entenderíamos quais seriam as próximas etapas do produto. Mas às 7h da manhã, o telefone tocou. Era o responsável pelo posto de gasolina piloto. Ele me ligou justamente para contar que o posto de gasolina estava **completamente parado** desde as 0h: o software parou de funcionar e bloqueou completamente o posto de gasolina.

Nosso software nunca havia sido testado com uma quantidade grande de abastecimentos. Os postos em Santo Domingo fazem muitos pequenos abastecimentos ao longo do dia (diferente daqui, onde enchemos o tanque de uma vez). O sistema não entendeu isso muito bem, e optou por bloquear as bombas de gasolina, para evitar fraude, já que não conseguia registrar as futuras compras.

O software fez com que o estabelecimento ficasse 12h parado, sem vender nada. Quanto será que isso custou ao dono do estabelecimento? Como será que foi a reação dele ao descobrir que o novo produto, no primeiro dia, causou tamanho estrago?

Os Estados Unidos estimam que bugs de software lhes custam aproximadamente 60 bilhões de dólares por ano [34]. O dinheiro que poderia estar sendo usado para erradicar a fome do planeta está sendo gasto em correções de software que não funcionam.

É incrível a quantidade de software que não funciona. Pergunte ao seu amigo não-técnico se ele já ficou irritado porque algum programa do seu dia a dia simplesmente parou de funcionar. Alguns bugs de software são inclusive famosos por todos: o foguete Ariane 5 explodiu por um erro de software; um hospital panamenho matou pacientes pois seu software para dosagem de remédios errou.

1.3 Por que não testamos?

Não há um desenvolvedor que não saiba que a solução para o problema é testar seus códigos. A pergunta é: **por que não testamos?**

Não testamos, porque testar sai caro. Imagine o sistema em que você trabalha hoje. Se uma pessoa precisasse testá-lo do começo ao fim, quanto tempo ela levaria? Semanas? Meses? Pagar um mês de uma pessoa a cada mudança feita no código (sim, os desenvolvedores também sabem que uma mudança em um trecho pode gerar problemas em outro) é simplesmente impossível.

Testar sai caro, no fim, porque estamos pagando "a pessoa" errada para fazer o trabalho. Acho muito interessante a quantidade de tempo que gastamos criando soluções tecnológicas para resolver problemas "dos outros". Por que não escrevemos programas que resolvam também os nossos problemas?

1.4 Testes automatizados e TDD

Uma maneira para conseguir testar o sistema todo de maneira constante e contínua a um preço justo é automatizando os testes. Ou seja, escrevendo um programa que testa o seu programa. Esse programa invocaria os comportamentos do seu sistema e garantiria que a saída é sempre a esperada.

Se isso fosse realmente viável, teríamos diversas vantagens. O teste executaria muito rápido (afinal, é uma máquina!). Se ele executa rápido, logo o rodaríamos constantemente. Se os rodarmos o tempo todo, descobriríamos os problemas mais cedo, diminuindo o custo que o bug geraria.

Um ponto que é sempre levantando em qualquer discussão sobre testes manuais versus testes automatizados é produtividade. O argumento mais comum é o de que agora a equipe de desenvolvimento gastará tempo escrevendo código de teste; antes ela só gastava tempo escrevendo código de produção. Portanto, essa equipe será menos produtiva.

A resposta para essa pergunta é: o que produtividade? Se produtividade for medida através do número de linhas de código de produção escritos por dia, talvez o desenvolvedor seja sim menos produtivo. Agora, se produtividade for a quantidade de linhas de código de produção sem defeitos escritos por dia, provavelmente o desenvolvedor será mais produtivo ao usar testes automatizados.

Além disso, se analisarmos o dia a dia de um desenvolvedor que faz testes manuais, podemos perceber a quantidade de tempo que ele gasta com teste. Geralmente o

desenvolvedor executa testes enquanto desenvolve o algoritmo completo. Ele escreve um pouco, roda o programa, e o programa falha. Nesse momento, o desenvolvedor entende o problema, corrige-o, e em seguida executa novamente o mesmo teste. Quantas vezes por dia o desenvolvedor executa o mesmo teste manual? O desenvolvedor que automatiza seus testes perde tempo apenas 1 vez com ele; nas próximas, ele simplesmente aperta um botão e vê a máquina executando o teste pra ele, de forma correta e rápida.

1.5 Conclusão

Minha família inteira é da área médica. Um jantar de fim de semana em casa parece mais um daqueles episódios de seriados médicos da televisão: pessoas discutindo casos e como resolvê-los. Apesar de entender praticamente nada sobre medicina, uma coisa me chama muito a atenção: o fanatismo deles por qualidade.

Um médico, ao longo de uma cirurgia, nunca abre mão de qualidade. Se o paciente falar para ele: "Doutor, o senhor poderia não lavar a mão e terminar a cirurgia 15 minutos mais cedo?", tenho certeza que o médico negaria na hora. Ele saberia que chegaria ao resultado final mais rápido, mas a chance de um problema é tão grande, que simplesmente não valeria a pena.

Em nossa área, vejo justamente o contrário. Qual desenvolvedor nunca escreveu um código de má qualidade de maneira consciente? Quem nunca escreveu uma "gambiarra"? Quem nunca colocou software em produção sem executar o mínimo suficiente de testes para tal?

Não há desculpas para não testar software. E a solução para que seus testes sejam sustentáveis é automatizando. Testar é divertido, aumenta a qualidade do seu produto, e pode ainda ajudá-lo a identificar trechos de código que foram mal escritos ou projetados (aqui entra a prática de TDD). É muita vantagem.

Ao longo do livro, espero convencê-lo de que testar é importante, e que na verdade é mais fácil do que parece.

1.6 Como tirar dúvidas?

Para facilitar a comunicação entre os leitores deste livro e interessados l no assunto, criei um grupo de discussão no Google Groups. Você pode se cadastrar em https://groups.google.com/forum/#!forum/tdd-no-mundo-real.

Toda e qualquer discussão, dúvida ou sugestão será bem vinda.

Não se esqueça também de participar do GUJ, o maior portal de desenvolvedores do Brasil: http://www.guj.com.br/

Lá você pode perguntar, responder, editar e dar pontos às dúvidas e respostas de outros desenvolvedores.

Capítulo 2

Testes de Unidade

2.1 O que é um teste de unidade?

Imagine-se passeando em uma loja virtual qualquer na web. Ao selecionar um produto, o sistema coloca-o no seu carrinho de compras. Ao finalizar a compra, o sistema fala com a operadora de cartão de crédito, retira o produto do estoque, dispara um evento para que a equipe de logística separe os produtos comprados e te envia um e-mail confirmando a compra.

O software que toma conta de tudo isso é complexo. Ele contém regras de negócio relacionadas ao carrinho de compras, ao pagamento, ao fechamento da compra. Mas, muito provavelmente, todo esse código não está implementado em apenas um único arquivo; esse sistema é composto por diversas pequenas classes, cada uma com sua tarefa específica.

Desenvolvedores, quando pensam em teste de software, geralmente imaginam um teste que cobre o sistema como um todo. Um teste de unidade não se preocupa com todo o sistema; ele está interessado apenas em saber se uma pequena parte do sistema funciona.

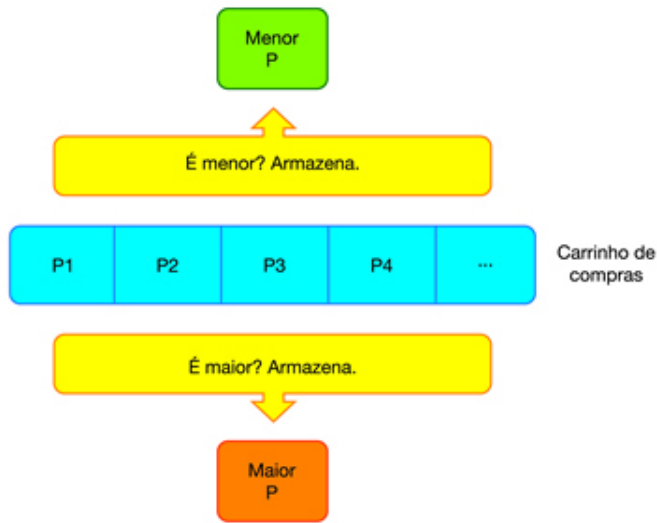
Um teste de unidade testa uma única unidade do nosso sistema. Geralmente, em sistemas orientados a objetos, essa unidade é a classe. Em nosso sistema de exemplo, muito provavelmente existem classes como "CarrinhoDeCompras", "Pedido", e assim por diante. A ideia é termos baterias de testes de unidade separadas para cada uma dessas classes; cada bateria preocupada apenas com a sua classe.

2.2 Preciso mesmo escrevê-los?

Essa mesma loja virtual precisa encontrar, dentro do seu carrinho de compras, os produtos de maior e menor valor. Um possível algoritmo para esse problema seria percorrer a lista de produtos no carrinho, comparar um a um, e guardar sempre a referência para o menor e o maior produto encontrado até então.

Em código, uma possível implementação seria:

```
public class MaiorEMenor {

    private Produto menor;
    private Produto maior;

    public void encontra(CarrinhoDeCompras carrinho) {
        for(Produto produto : carrinho.getProdutos()) {
            if(menor == null || produto.getValor() < menor.getValor()) {
                menor = produto;
            }
            else if (maior == null ||
                    produto.getValor() > maior.getValor()) {
                maior = produto;
            }
        }
    }

    public Produto getMenor() {
        return menor;
    }

    public Produto getMaior() {
        return maior;
    }

}
```

Veja o método `encontra()`. Ele recebe um `CarrinhoDeCompras` e percorre a lista de produtos, comparando sempre o produto corrente com o "menor e maior de todos". Ao final, temos no atributo `maior` e `menor` os produtos desejados. A figura abaixo mostra como o algoritmo funciona:

Para exemplificar o uso dessa classe, veja o código abaixo:

```
public class TestaMaiorEMenor {
    public static void main(String[] args) {
        CarrinhoDeCompras carrinho = new CarrinhoDeCompras();
        carrinho.adiciona(new Produto("Liquidificador", 250.0));
        carrinho.adiciona(new Produto("Geladeira", 450.0));
        carrinho.adiciona(new Produto("Jogo de pratos", 70.0));

        MaiorEMenor algoritmo = new MaiorEMenor();
        algoritmo.encontra(carrinho);

        System.out.println("O menor produto: " +
            algoritmo.getMenor().getNome());
        System.out.println("O maior produto: " +
            algoritmo.getMaior().getNome());
    }
}
```

O carrinho contém três produtos: liquidificador, geladeira e jogo de pratos. É fácil perceber que o jogo de pratos é o produto mais barato (R$ 70,00), enquanto que a geladeira é o mais caro (R$ 450,00). A saída do programa é exatamente igual à esperada:

```
O menor produto: Jogo de pratos
O maior produto: Geladeira
```

Apesar de aparentemente funcionar, se esse código for para produção, a loja virtual terá problemas.

2.3 O Primeiro Teste de Unidade

A classe `MaiorEMenor` respondeu corretamente ao teste feito acima, mas ainda não é possível dizer se ela realmente funciona para outros cenários. Observe o código abaixo:

```
public class TestaMaiorEMenor {
    public static void main(String[] args) {
        CarrinhoDeCompras carrinho = new CarrinhoDeCompras();
        carrinho.adiciona(new Produto("Geladeira", 450.0));
        carrinho.adiciona(new Produto("Liquidificador", 250.0));
        carrinho.adiciona(new Produto("Jogo de pratos", 70.0));

        MaiorEMenor algoritmo = new MaiorEMenor();
        algoritmo.encontra(carrinho);

        System.out.println("O menor produto: " +
            algoritmo.getMenor().getNome());
        System.out.println("O maior produto: " +
            algoritmo.getMaior().getNome());
    }
}
```

O código acima não é tão diferente do anterior. Os produtos, bem como os valores, são os mesmos; apenas a ordem em que eles são inseridos no carrinho foi trocada. Espera-se então que o programa produza a mesma saída. Mas, ao executá-lo, a seguinte saída é gerada:

```
O menor produto: Jogo de pratos
Exception in thread "main" java.lang.NullPointerException
    at TestaMaiorEMenor.main(TestaMaiorEMenor.java:12)
```

Problema! Essa não era a saída esperada! Agora está claro que a classe `MaiorEMenor` não funciona bem para todos os cenários. Se os produtos, por algum motivo, forem adicionados no carrinho em ordem decrescente, a classe não consegue calcular corretamente.

Uma pergunta importante para o momento é: será que o desenvolvedor, ao escrever a classe, perceberia esse bug? Será que ele faria todos os testes necessários para garantir que a classe realmente funciona?

Como discutido no capítulo anterior, equipes de software tendem a não testar software, pois o teste leva tempo e, por consequência, dinheiro. Na prática, equipes acabam por executar poucos testes ralos, que garantem apenas o cenário feliz e mais comum. Todos os problemas da falta de testes, que já foram discutidos anteriormente, agora fazem sentido. Imagine se a loja virtual colocasse esse código em produção. Quantas compras seriam perdidas por causa desse problema?

Para diminuir a quantidade de bugs levados para o ambiente de produção, é necessário testar o código constantemente. Idealmente, a cada alteração feita, todo o sistema deve ser testado por inteiro novamente. Mas isso deve ser feito de maneira sustentável: é impraticável pedir para que seres humanos testem o sistema inteiro a cada alteração feita por um desenvolvedor.

A solução para o problema é fazer com que a máquina teste o software. A máquina executará o teste rapidamente e sem custo, e o desenvolvedor não gastaria mais tempo executando testes manuais, mas sim apenas em evoluir o sistema.

Escrever um teste automatizado não é tarefa tão árdua. Ele, na verdade, se parece muito com um teste manual. Imagine um desenvolvedor que deva testar o comportamento do carrinho de compras da loja virtual quando existem dois produtos lá cadastrados: primeiramente, ele "clicaria em comprar em dois produtos", em seguida "iria para o carrinho de compras", e por fim, verificaria "a quantidade de itens no carrinho (deve ser 2)" e o "o valor total do carrinho (que deve ser a soma dos dois produtos adicionados anteriormente)".

Ou seja, de forma generalizada, o desenvolvedor primeiro pensa em um cenário (dois produtos comprados), depois executa uma ação (vai ao carrinho de compras), e por fim, valida a saída (vê a quantidade de itens e o valor total do carrinho). A figura abaixo mostra os passos que um testador geralmente faz quando deseja testar uma funcionalidade.

Um teste automatizado é similar. Ele descreve um cenário, executa uma ação e valida uma saída. A diferença é que quem fará tudo isso será a máquina, sem qualquer intervenção humana.

De certa forma, um teste automatizado já foi escrito neste capítulo. Veja o código abaixo, escrito para testar superficialmente a classe `MaiorEMenor`:

```
public class TestaMaiorEMenor {
    public static void main(String[] args) {
        CarrinhoDeCompras carrinho = new CarrinhoDeCompras();
        carrinho.adiciona(new Produto("Geladeira", 450.0));
        carrinho.adiciona(new Produto("Liquidificador", 250.0));
        carrinho.adiciona(new Produto("Jogo de pratos", 70.0));

        MaiorEMenor algoritmo = new MaiorEMenor();
        algoritmo.encontra(carrinho);

        System.out.println("O menor produto: " +
                                algoritmo.getMenor().getNome());
        System.out.println("O maior produto: " +
                                algoritmo.getMaior().getNome());
    }
}
```

Veja que esse código contém, de forma automatizada, boa parte do que foi descrito em relação ao teste manual: ele monta um cenário (um carrinho de compras com 3 produtos), executa uma ação (invoca o método `encontra()`), e valida a saída (imprime o maior e o menor produto).

E o melhor: uma máquina faz (quase) tudo isso. Ao rodar o código acima, a máquina monta o cenário e executa a ação sem qualquer intervenção humana. Mas uma ação humana ainda é requerida para descobrir se o comportamento executou de acordo. Um humano precisa ver a saída e conferir com o resultado esperado.

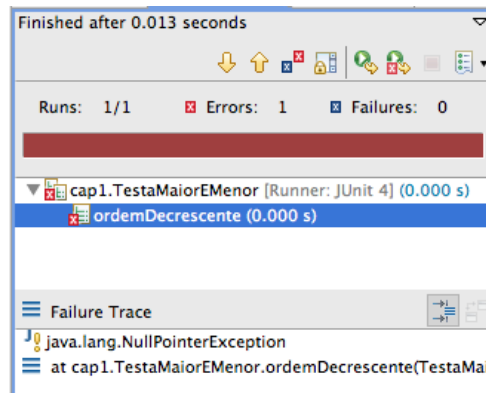
É preciso que o próprio teste faça a validação e informe o desenvolvedor caso o resultado não seja o esperado. Para melhorar o código acima, agora só introduzindo um framework de teste automatizado. Esse framework nos daria um relatório mais

detalhado dos testes que foram executados com sucesso e, mais importante, dos testes que falharam, trazendo o nome do teste e a linha que apresentaram problemas. Neste livro, faremos uso do JUnit [1], o framework de testes de unidade mais popular do mundo Java.

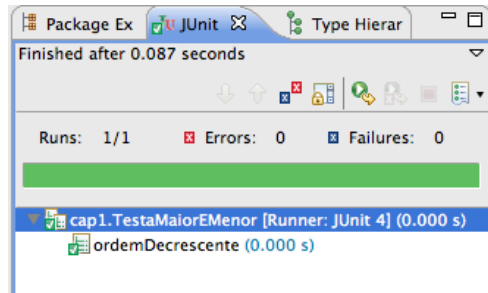
O JUnit possui um plugin para Eclipse, que mostra uma lista de todos os testes executados, pintando-os de verde em caso de sucesso, ou vermelho em caso de falha. Ao clicar em um teste vermelho, a ferramenta ainda apresenta a linha que falhou, o resultado que era esperado e o resultado devolvido pelo método.

Para converter o código acima em um teste que o JUnit entenda, não é necessário muito trabalho. A única parte que ainda não é feita 100% pela máquina é a terceira parte do teste: a validação. É justamente ali que invocaremos os métodos do JUnit. Eles que farão a comparação do resultado esperado com o calculado. No JUnit, o método para isso é o `Assert.assertEquals()`:

```
import org.junit.Assert;
import org.junit.Test;

public class TestaMaiorEMenor {

    @Test
    public void ordemDecrescente() {
        CarrinhoDeCompras carrinho = new CarrinhoDeCompras();
        carrinho.adiciona(new Produto("Geladeira", 450.0));
        carrinho.adiciona(new Produto("Liquidificador", 250.0));
        carrinho.adiciona(new Produto("Jogo de pratos", 70.0));

        MaiorEMenor algoritmo = new MaiorEMenor();
        algoritmo.encontra(carrinho);

        Assert.assertEquals("Jogo de pratos",
            algoritmo.getMenor().getNome());
        Assert.assertEquals("Geladeira",
            algoritmo.getMaior().getNome());
    }
}
```

Pronto. Esse código acima é executado pelo JUnit. O teste, como bem sabemos, falhará:

Para fazer o teste passar, é necessário corrigir o bug na classe de produção. O que faz o código falhar é justamente a presença do `else` (ele fazia com que o código nunca passasse pelo segundo `if`, que verificava justamente o maior elemento). Ao removê-lo, o teste passa:

```
public class MaiorEMenor {

    private Produto menor;
    private Produto maior;

    public void encontra(CarrinhoDeCompras carrinho) {
        for(Produto produto : carrinho.getProdutos()) {
            if(menor == null || produto.getValor() < menor.getValor()) {
                menor = produto;
            }
            if (maior == null || produto.getValor() > maior.getValor()){
                maior = produto;
            }
        }
    }

    public Produto getMenor() {
        return menor;
    }

    public Produto getMaior() {
        return maior;
    }
```

```
}
```

Repare também no tempo que a máquina levou para executar o teste: 0.007 segundos. Mais rápido do que qualquer ser humano faria. Isso possibilita ao desenvolvedor executar esse teste diversas vezes ao longo do seu dia de trabalho.

E o melhor: se algum dia um outro desenvolvedor alterar esse código, ele poderá executar a bateria de testes automatizados existente e descobrir se a sua alteração fez alguma funcionalidade que já funcionava anteriormente parar de funcionar. Isso é conhecido como **testes de regressão**. Eles garantem que o sistema não regrediu.

Meu primeiro teste automatizado

Há muitos anos atrás, trabalhava em uma pequena consultoria de desenvolvimento de software em São Paulo. E, como em toda consultoria, precisávamos anotar nossas horas de trabalho em cada um dos projetos da empresa, a fim de cobrar corretamente cada um dos clientes.

Resolvi, por conta própria, criar um simples projeto para controle de horas de trabalho (eu precisava de um motivo para experimentar todas as coisas que estava aprendendo naquele momento e, dentre elas, testes automatizados). O sistema era bem simples: cadastro de funcionários, cadastro de projetos, ligação entre um funcionário e um projeto e cadastro de horas em um projeto.

Ao final da implementação, apresentei o sistema ao diretor da empresa. Ele adorou, e me sugeriu a implementação de uma simples regra de negócio: se o número total de horas colocadas no projeto ultrapasse um determinado limite, nenhum funcionário poderia adicionar mais horas nele.

A implementação era bem trivial: bastava fazer um `if`. Lembro-me que implementei a regra de negócio e então escrevi o teste de unidade para garantir que havia implementado corretamente. O teste que acabei de escrever ficou verde, mas três ou quatro testes que já existiam ficaram vermelhos. A única linha de código que escrevi fez com que funcionalidades que já funcionavam, parassem de trabalhar corretamente.

Naquele momento me veio a cabeça todas as vezes que escrevi uma ou duas linhas de código, aparentemente simples, mas que poderiam ter quebrado o sistema. Percebi então a grande vantagem da bateria de testes automatizados: segurança. Percebi o como é fácil quebrar código, e como é difícil perceber isso sem testes automatizados. Daquele dia em diante, penso duas vezes antes de escrever código sem teste.

2.4 Continuando a testar

Ainda são necessários muitos testes para garantir que a classe `MaiorEMenor` funcione corretamente. Nesse momento, o único cenário testado são produtos em ordem decrescente. Muitos outros cenários precisam ser testados. Dentre eles:

- Produtos com valores em ordem crescente
- Produtos com valores em ordem variada
- Um único produto no carrinho

Conhecendo o código da classe `MaiorEMenor`, é possível prever que os cenários acima funcionarão corretamente. Escrever testes automatizados para os cenários acima pode parecer inútil nesse momento.

Entretanto, é importante lembrar que em códigos reais, muitos desenvolvedores fazem alterações nele. Para o desenvolvedor que implementa a classe, o código dela é bem natural e simples de ser entendido. Mas para os futuros desenvolvedores que a alterarão, esse código pode não parecer tão simples. É necessário então prover segurança para eles nesse momento. Portanto, apesar de alguns testes parecerem desnecessários nesse momento, eles garantirão que a evolução dessa classe será feita com qualidade.

Apenas para exemplificar mais ainda, vamos escrever mais um teste, dessa vez para um carrinho com apenas 1 produto. Repare que nesse cenário, é esperado que o maior e menor produtos seja o mesmo:

```java
import org.junit.Assert;
import org.junit.Test;

public class TestaMaiorEMenor {

    @Test
    public void apenasUmProduto() {
        CarrinhoDeCompras carrinho = new CarrinhoDeCompras();
        carrinho.adiciona(new Produto("Geladeira", 450.0));

        MaiorEMenor algoritmo = new MaiorEMenor();
        algoritmo.encontra(carrinho);

        Assert.assertEquals("Geladeira",
            algoritmo.getMenor().getNome());
        Assert.assertEquals("Geladeira",
            algoritmo.getMaior().getNome());
    }
}
```

Comparando só o nome?

Ao invés de comparar apenas o nome do produto, você poderia comparar o objeto inteiro: `Assert.assertEquals(produto, algoritmo.getMenor());`, por exemplo. Mas, para isso, o método `equals()` deve estar implementado na classe `Produto`.

Em certos casos, essa solução pode ser mais interessante, afinal você comparará o objeto inteiro e não só apenas um atributo.

2.5 Conclusão

Desenvolvedores gastam toda sua vida automatizando processos de outras áreas de negócio, criando sistemas para RHs, controle financeiro, entre outros, com o intuito de facilitar a vida daqueles profissionais. Por que não criar software que automatize o seu próprio ciclo de trabalho?

Testes automatizados são fundamentais para um desenvolvimento de qualidade, e é obrigação de todo desenvolvedor escrevê-los. Sua existência traz diversos benefícios para o software, como o aumento da qualidade e a diminuição de bugs em produção. Nos próximos capítulos, discutiremos sobre como aumentar ainda mais o *feedback* que os testes nos dão sobre a qualidade do software.

Capítulo 3

Introdução ao Test-Driven Development

Desenvolvedores (ou qualquer outro papel que é executado dentro de uma equipe de software) estão muito acostumados com o "processo tradicional" de desenvolvimento: primeiro a implementação e depois o teste. Entretanto, uma pergunta interessante é: *Será que é possível inverter, ou seja, testar primeiro e depois implementar? E mais importante, faz algum sentido?*

Para responder essa pergunta, ao longo deste capítulo desenvolveremos um simples algoritmo matemático, mas dessa vez invertendo o ciclo de desenvolvimento: o teste será escrito antes da implementação. Ao final, discutiremos as vantagens e desvantagens dessa abordagem.

3.1 O problema dos números romanos

Numerais romanos foram criados na Roma Antiga e eles foram utilizados em todo o seu império. Os números eram representados por sete diferentes símbolos, listados na tabela a seguir.

- I, unus, 1, (um)
- V, quinque, 5 (cinco)
- X, decem, 10 (dez)
- L, quinquaginta, 50 (cinquenta)
- C, centum, 100 (cem)
- D, quingenti, 500 (quinhentos)
- M, mille, 1.000 (mil)

Para representar outros números, os romanos combinavam estes símbolos, começando do algarismo de maior valor e seguindo a regra:

- Algarismos de menor ou igual valor à direita são somados ao algarismo de maior valor;
- Algarismos de menor valor à esquerda são subtraídos do algarismo de maior valor.

Por exemplo, XV representa 15 (10 + 5) e o número XXVIII representa 28 (10 + 10 + 5 + 1 + 1 + 1). Há ainda uma outra regra: nenhum símbolo pode ser repetido lado a lado por mais de 3 vezes. Por exemplo, o número 4 é representado pelo número IV (5 - 1) e não pelo número IIII.

Existem outras regras (especialmente para números maiores, que podem ser lidas aqui [33]), mas em linhas gerais, este é o problema a ser resolvido. Dado um numeral romano, o programa deve convertê-lo para o número inteiro correspondente.

3.2 O primeiro teste

Conhecendo o problema dos numerais romanos, é possível levantar os diferentes cenários que precisam ser aceitos pelo algoritmo: um símbolo, dois símbolos iguais, três símbolos iguais, dois símbolos diferentes do maior para o menor, quatro símbolos dois a dois, e assim por diante. Dado todos estes cenários, uns mais simples que os outros, começaremos pelo mais simples: um único símbolo.

Começando pelo teste "deve entender o símbolo I". A classe responsável pela conversão pode ser chamada, por exemplo, de *ConversorDeNumeroRomano*, e o método *converte()*, recebendo uma String com o numeral romano e devolvendo o valor inteiro representado por aquele número:

```
public class ConversorDeNumeroRomanoTest {

    @Test
    public void deveEntenderOSimboloI() {
        ConversorDeNumeroRomano romano = new ConversorDeNumeroRomano();
        int numero = romano.converte("I");
        assertEquals(1, numero);
    }

}
```

> **Import estático**
>
> Repare que aqui utilizamos `assertEquals()` diretamente. Isso é possível graças ao import estático do Java 5.0. Basta importar `import static org.junit.Assert.*` e todos os métodos estáticos dessa classe estarão disponíveis sem a necessidade de colocar o nome dela.

Veja que nesse momento, esse código não compila; a classe *ConversorDeNumeroRomano*, bem como o método *converte()* não existem. Para resolver o erro de compilação, é necessário criar classe, mesmo que sem uma implementação real:

```
public class ConversorDeNumeroRomano {

    public int converte(String numeroEmRomano) {
        return 0;
```

```
    }

}
```

De volta ao teste, ele agora compila. Ao executá-lo, o teste falha. Mas não há problema; isso já era esperado. Para fazê-lo passar, introduziremos ainda uma segunda regra: **o código escrito deve ser sempre o mais simples possível**.

Com essa regra em mente, o código mais simples que fará o teste passar é fazer simplesmente o método *converte()* retornar o número 1:

```
public int converte(String numeroEmRomano) {
    return 1;
}
```

Desenvolvedores, muito provavelmente, não ficarão felizes com essa implementação, afinal ela funciona apenas para um caso. Mas isso não é problema, afinal a implementação não está pronta; ainda estamos trabalhando nela.

Um próximo cenário seria o símbolo V. Nesse caso, o algoritmo deve retornar 5. Novamente começando pelo teste:

```
@Test
public void deveEntenderOSimboloV() {
    ConversorDeNumeroRomano romano = new ConversorDeNumeroRomano();
    int numero = romano.converte("V");
    assertEquals(5, numero);
}
```

Esse teste também falha. Novamente faremos a implementação mais simples que resolverá o problema. Podemos, por exemplo, fazer com que o método *converte()* verifique o conteúdo do número a ser convertido: se o valor for "I", o método retorna 1; se o valor for "V", o método retorna 5:

```
public int converte(String numeroEmRomano) {
    if(numeroEmRomano.equals("I")) return 1;
    else if(numeroEmRomano.equals("V")) return 5;

    return 0;
}
```

Os testes passam (os dois que temos). Poderíamos repetir o mesmo teste e a mesma implementação para os símbolos que faltam (X, L, C, M, ...). Mas nesse mo-

mento, já temos uma primeira definição sobre nosso algoritmo: quando o numeral romano possui apenas um símbolo, basta devolvermos o inteiro associado a ele.

Ao invés de escrever um monte de `ifs` para cada símbolo, é possível usar um *switch*, que corresponde melhor ao nosso cenário. Melhorando ainda mais, ao invés do *switch*, é possível guardar os símbolos em uma tabela, ou no Java, em um mapa entre o algarismo e o inteiro correspondente à ele. Sabendo disso, vamos nesse momento alterar o código para refletir a solução:

```
public class ConversorDeNumeroRomano {

    private static Map<String, Integer> tabela =
        new HashMap<String, Integer>() {{
        put("I", 1);
        put("V", 5);
        put("X", 10);
        put("L", 50);
        put("C", 100);
        put("D", 500);
        put("M", 1000);
    }};

    public int converte(String numeroEmRomano) {
        return tabela.get(numeroEmRomano);
    }

}
```

Ambos os testes continuam passando. Passaremos agora para um segundo cenário: dois símbolos em sequência, como por exemplo, "II" ou "XX". Começando novamente pelo teste, temos:

```
@Test
public void deveEntenderDoisSimbolosComoII() {
    ConversorDeNumeroRomano romano = new ConversorDeNumeroRomano();
    int numero = romano.converte("II");
    assertEquals(2, numero);
}
```

Para fazer o teste passar de maneira simples, é possível simplesmente adicionar os símbolos "II" na tabela:

```
private static Map<String, Integer> tabela =
    new HashMap<String, Integer>() {{
    put("I", 1);

    put("II", 2);

    put("V", 5);
    put("X", 10);
    put("L", 50);
    put("C", 100);
    put("D", 500);
    put("M", 1000);
}};
```

O teste passa. Mas, apesar de simples, essa não parece uma boa ideia de implementação: seria necessário incluir todos os possíveis símbolos nessa tabela, o que não faz sentido.

É hora de refatorar esse código novamente. Uma possível solução seria iterar em cada um dos símbolos no numeral romano e acumular seu valor; ao final, retorna o valor acumulado. Para isso, é necessário mudar a `tabela` para guardar somente os símbolos principais da numeração romana. Uma possível implementação deste algoritmo seria:

```
private static Map<Character, Integer> tabela =
    new HashMap<Character, Integer>() {{
    put('I', 1);
    put('V', 5);
    put('X', 10);
    put('L', 50);
    put('C', 100);
    put('D', 500);
    put('M', 1000);
}};

public int converte(String numeroEmRomano) {
    int acumulador = 0;
    for(int i = 0; i < numeroEmRomano.length(); i++) {
        acumulador += tabela.get(numeroEmRomano.charAt(i));
    }
    return acumulador;
}
```

De String para Char?

Repare que o tipo da chave no mapa mudou de String para Character. Como o algoritmo captura letra por letra da variável `numeroEmRomano`, usando o método `charAt()`, que devolve um char, a busca do símbolo fica mais direta.

Os três testes continuam passando. E, dessa forma, resolvemos o problema de dois símbolos iguais em seguida. O próximo cenário são quatro símbolos, dois a dois, como por exemplo, "XXII", que deve resultar em 22. O teste:

```
@Test
public void deveEntenderQuatroSimbolosDoisADoisComoXXII() {
    ConversorDeNumeroRomano romano = new ConversorDeNumeroRomano();
    int numero = romano.converte("XXII");
    assertEquals(22, numero);
}
```

Esse teste já passa sem que precisemos fazer qualquer alteração. O algoritmo existente até então já resolve o problema. Vamos então para o próximo cenário: números como "IV" ou "IX", onde não basta apenas somar os símbolos existentes.

Novamente, começando pelo teste:

```
@Test
public void deveEntenderNumerosComoIX() {
    ConversorDeNumeroRomano romano = new ConversorDeNumeroRomano();
    int numero = romano.converte("IX");
    assertEquals(9, numero);
}
```

Para fazer esse teste passar, é necessário pensar um pouco melhor sobre o problema. Repare que os símbolos em um numeral romano, da direita para a esquerda, sempre crescem. Quando um número a esquerda é menor do que seu vizinho a direita, esse número deve então ser subtraído ao invés de somado no acumulador. Esse algoritmo, em código:

```
public int converte(String numeroEmRomano) {
    int acumulador = 0;
    int ultimoVizinhoDaDireita = 0;
```

```
    for(int i = numeroEmRomano.length() - 1; i >= 0; i--) {

        // pega o inteiro referente ao simbolo atual
        int atual = tabela.get(numeroEmRomano.charAt(i));

        // se o da direita for menor, o multiplicaremos
        // por -1 para torná-lo negativo
        int multiplicador = 1;
        if(atual < ultimoVizinhoDaDireita) multiplicador = -1;

        acumulador +=
            tabela.get(numeroEmRomano.charAt(i)) * multiplicador;

        // atualiza o vizinho da direita
        ultimoVizinhoDaDireita = atual;
    }
    return acumulador;
}
```

Os testes agora passam. Mas veja, existe código repetido ali. Na linha em que o acumulador é somado com o valor atual, *tabela.get(numeroEmRomano.charAt(i))* pode ser substituído pela variável *atual*. Melhorando o código, temos:

```
acumulador += atual * multiplicador;
```

A refatoração foi feita com sucesso, já que os testes continuam passando. Ao próximo cenário: numeral romano que contenha tanto números "invertidos", como "IV", e dois símbolos lado a lado, como "XX". Por exemplo, o número "XXIV", que representa o número 24.

Começando pelo teste:

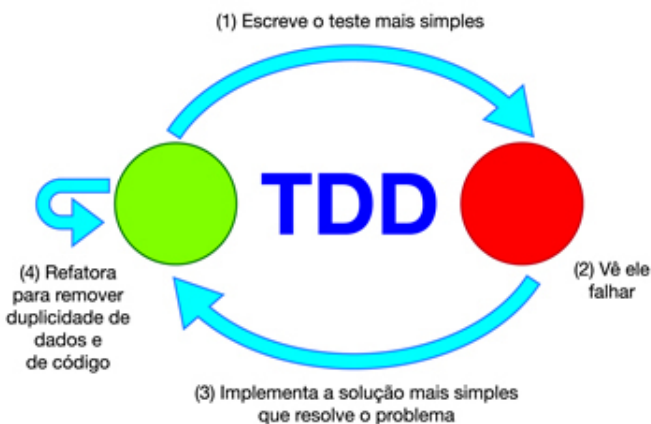
```
@Test
public void deveEntenderNumerosComplexosComoXXIV() {
    ConversorDeNumeroRomano romano = new ConversorDeNumeroRomano();
    int numero = romano.converte("XXIV");
    assertEquals(24, numero);
}
```

Esse teste já passa; o algoritmo criado até então já atende o cenário do teste. Ainda há outros cenários que poderiam ser testados, mas já fizemos o suficiente para discutir sobre o assunto.

3.3 Refletindo sobre o assunto

De maneira mais abstrata, o ciclo que foi repetido ao longo do processo de desenvolvimento da classe acima foi:

- Escrevemos um teste de unidade para uma nova funcionalidade;
- Vimos o teste falhar;
- Implementamos o código mais simples para resolver o problema;
- Vimos o teste passar;
- Melhoramos (refatoramos) nosso código quando necessário.

Esse ciclo de desenvolvimento é conhecido por **Test-Driven Development (TDD)**, ou, **Desenvolvimento Guiado pelos Testes**. A ideia é simples: o desenvolvedor deve começar a implementação pelo teste e, deve o tempo todo, fazer de tudo para que seu código fique simples e com qualidade.

Esse ciclo é também conhecido como ciclo **vermelho-verde-refatora** (ou red-green-refactor). O primeiro passo é escrever um teste que falha. A cor vermelha representa esse teste falhando. Em seguida, fazemos ele passar (a cor verde representa ele passando). Por fim, refatoramos para melhorar o código que escrevemos.

3.4 Quais as vantagens?

Muitos praticantes de TDD afirmam que executar esse ciclo pode ser muito vantajoso para o processo de desenvolvimento. Alguns deles:

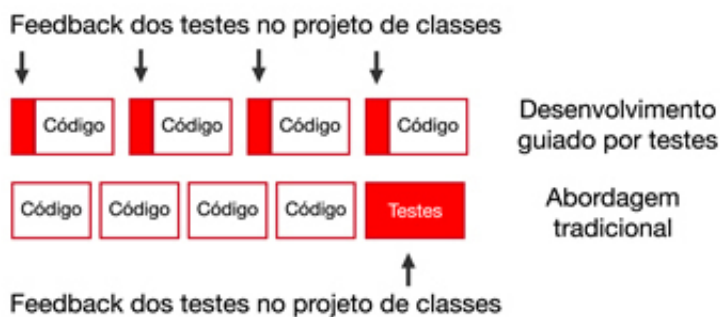
- **Foco no teste e não na implementação**. Ao começar pelo teste, o programador consegue pensar somente no que a classe **deve** fazer, e esquece por um momento da implementação. Isso o ajuda a pensar em melhores cenários de teste para a classe sob desenvolvimento.
- **Código nasce testado**. Se o programador pratica o ciclo corretamente, isso então implica em que todo o código de produção escrito possui ao menos um teste de unidade verificando que ele funciona corretamente.
- **Simplicidade**. Ao buscar pelo código mais simples constantemente, o desenvolvedor acaba por fugir de soluções complexas, comuns em todos os sistemas. O praticante de TDD escreve código que apenas resolve os problemas que estão representados por um teste de unidade. Quantas vezes o desenvolvedor não escreve código desnecessariamente complexo?
- **Melhor reflexão sobre o design da classe**. No cenário tradicional, muitas vezes a falta de coesão ou o excesso de acoplamento é causado muitas vezes pelo desenvolvedor que só pensa na implementação e acaba esquecendo como a classe vai funcionar perante o todo. Ao começar pelo teste, o desenvolvedor pensa sobre como sua classe deverá se comportar perante as outras classes do sistema. O teste atua como o primeiro cliente da classe que está sendo escrita. Nele, o desenvolvedor toma decisões como o nome da classe, os seus métodos, parâmetros, tipos de retorno, e etc. No fim, todas elas são decisões de design e, quando o desenvolvedor consegue observar com atenção o código do teste, seu design de classes pode crescer muito em qualidade.

Todos estes pontos serão melhor descritos e aprofundados no capítulo a seguir. Nesse momento, foque nestas vantagens. Uma pergunta que pode vir a cabeça é: *"No modelo tradicional, onde os testes são escritos depois, o desenvolvedor não tem os mesmos benefícios?"*

A resposta é sim. Mesmo desenvolvedores que escrevem testes depois podem obter as mesmas vantagens. Um desenvolvedor pode perceber que está com dificuldades de escrever um teste e descobrir que o design da classe que implementou tem

problemas; ele pode também conseguir abstrair a implementação e escrever bons cenários de teste.

Mas há uma diferença que faz toda a diferença: a quantidade de vezes que um programador praticante de TDD recebe *feedback* sobre esses pontos e a quantidade que um programador que não pratica TDD recebe.

Veja a figura abaixo. O praticante de TDD escreve um pouco de testes, um pouco de implementação e recebe *feedback*. Isso acontece ao longo do desenvolvimento de maneira frequente. Já um programador que não pratica TDD espera um tempo (às vezes longo demais) para obter o mesmo *feedback*.

Ao receber *feedback* desde cedo, o programador pode melhorar o código e corrigir problemas a um custo menor do que o programador que recebeu a mesma mensagem muito tempo depois. Todo programador sabe que alterar o código que ele acabara de escrever é muito mais fácil e rápido do que alterar o código escrito 3 dias atrás.

No fim, TDD apenas maximiza a quantidade de *feedback* sobre o código que está sendo produzido, fazendo o programador perceber os problemas antecipadamente e, por consequência, diminuindo os custos de manutenção e melhorando o código.

Desenvolvedores e a busca pela complexidade

Acho impressionante como nós desenvolvedores somos atraídos por complexidade. Gostamos de problemas difíceis, que nos desafiem. Lembro-me que no começo de carreira, quando pegava alguma funcionalidade simples de ser implementada, eu mesmo "aumentava a funcionalidade" somente para torná-la interessante do ponto de vista técnico. Ou seja, um simples relatório se tornava um relatório complexo, com diversos filtros e que podia ser exportado para muitos formatos diferentes.

Aprender a ser simples foi difícil. O TDD, de certa forma, me ajudou nisso. Ao pensar de pouco em pouco, e implementar somente o necessário para resolver o problema naquele momento, comecei a perceber que poderia gastar meu tempo implementando funcionalidades que realmente agregariam valor para o usuário final.

Portanto, não ache que "ser simples" é fácil. Lembre-se que somos atraídos por complexidade.

3.5 Um pouco da história de TDD

TDD ficou bastante popular após a publicação do livro *TDD: By Example*, do Kent Beck, em 2002. O próprio Kent afirma que TDD não foi uma ideia totalmente original. Ele conta que em algum momento de sua vida, que leu em algum dos livros de seu pai (que também era programador), sobre uma técnica de testes mais antiga, onde o programador colocava na fita o valor que ele esperava daquele programa, e então o programador desenvolvia até chegar naquele valor.

Ele próprio conta que achou a ideia estúpida. Qual o sentido de escrever um teste que falha? Mas resolveu experimentar. Após a experiência, ele disse que "as pessoas sempre falavam pra ele conseguir separar o que o programa deve fazer, da sua implementação final, mas que ele não sabia como fazer, até aquele momento em que resolveu escrever o teste antes."

Daquele momento em diante, Kent Beck continuou a trabalhar na ideia. Em 1994, ele escreveu o seu primeiro framework de testes de unidade, o SUnit (para Smalltalk). Em 1995, ele apresentou TDD pela primeira vez na OOPSLA (conferência muito famosa da área de computação, já que muitas novidades tendem a aparecer lá).

Já em 2000, o JUnit surgiu e Kent Beck, junto com Erich Gamma, publicou o

artigo chamado de "Test Infected" [2], que mostrava as vantagens de se ter testes automatizados e como isso pode ser viciante.

Finalmente em 2002, Kent Beck lançou seu livro sobre isso, e desde então a prática tem se tornado cada vez mais popular entre os desenvolvedores.

A história mais completa pode ser vista na Wiki da C2 [3] ou na palestra do Steve Freeman e Michael Feathers na QCON de 2009 [16].

> **Ferreira fala**
>
> Duas histórias muito conhecidas cercam Kent Beck e o nascimento dos primeiros frameworks de testes unitários. O SUnit, para Smalltalk, não era um código distribuído em arquivos da maneira usual. Beck na época atuava como consultor e tinha o hábito de recriar o framework junto com os programadores de cada cliente. A criação do JUnit também é legendária: a primeira versão foi desenvolvida numa sessão de programação pareada entre o Kent Beck e o Erich Gamma em um voo Zurique-Atlanta.

3.6 Conclusão

Neste capítulo, apresentamos TDD e mostramos algumas das vantagens que o programador obtém ao utilizá-la no seu dia a dia. Entretanto, é possível melhorar ainda mais a maneira na qual o programador faz uso da prática.

Somente praticando TDD com esse simples exemplo, é possível levantar diversas questões, como:

- Muitos cenários foram deixados para trás, como por exemplo, a conversão de números como "CC", "MM", e etc. Eles devem ser escritos?
- O inverso da pergunta anterior: testes para "I", "V", "X" devem ser considerados diferentes ou repetidos?
- E quanto a repetição de código dentro de cada teste? É possível melhorar a qualidade de código do próprio teste?
- A ideia de sempre fazer o teste passar da maneira mais simples possível faz sentido? Deve ser aplicado 100% do tempo?

- Existe alguma regra para dar nomes aos testes?
- Qual o momento ideal para refatorar o código?
- Se durante a implementação, um teste antigo, que passava, quebra? O que devo fazer?

Para que um desenvolvedor faça uso de TDD de maneira profissional e produtiva, estes e outros pontos devem ser discutidos e clarificados. Nos próximos capítulos, responderemos a cada uma dessas perguntas, usando como exemplo um sistema razoavelmente complexo muito similar aos encontrados no mundo real.

Capítulo 4

Simplicidade e Baby Steps

No capítulo anterior, apresentamos TDD em um exemplo complexo do ponto de vista algorítmico, mas bem controlado. Mesmo assim, o capítulo anterior foi muito útil e serviu para mostrar os passos de um desenvolvedor que pratica TDD.

O mundo real é mais complexo do que isso; algoritmos matemáticos são combinados e interagem com entidades que vem do banco de dados e são apresentados para o usuário através de alguma interface. De agora em diante, os exemplos a serem trabalhados serão outros e, com certeza, mais parecido com a realidade. No mundo real, as coisas evoluem e mudam rapidamente. Portanto, nos próximos exemplos, sempre imagine que eles ficarão mais complexos e evoluirão.

4.1 O Problema do Cálculo de Salário

Suponha que uma empresa precise calcular o salário do funcionário e seus descontos. Para calcular esse desconto, a empresa leva em consideração o salário atual e o cargo do funcionário. Vamos representar funcionários e cargos da seguinte maneira:

```
public enum Cargo {
    DESENVOLVEDOR,
    DBA,
    TESTADOR
}

public class Funcionario {

    private String nome;
    private double salario;
    private Cargo cargo;

    public Funcionario(String nome, double salario, Cargo cargo) {
        this.nome = nome;
        this.salario = salario;
        this.cargo = cargo;
    }

    public String getNome() {
        return nome;
    }

    public double getSalario() {
        return salario;
    }

    public Cargo getCargo() {
        return cargo;
    }

}
```

Repare que um funcionário possui nome, salário e cargo. O cargo é representado por uma enumeração. Nesse momento, a enumeração contém apenas desenvolvedor, testador e DBA. Em uma situação real, essa enumeração seria maior ainda; mas com esses 3 já gera uma boa discussão.

As regras de negócio são as seguintes:

- Desenvolvedores possuem 20% de desconto caso seu salário seja maior do que R$ 3000,0. Caso contrário, o desconto é de 10%.

- DBAs e testadores possuem desconto de 25% se seus salários forem maiores do que R$ 2500,0. 15%, em caso contrário.

4.2 Implementando da maneira mais simples possível

É hora de implementar essas regras. Uma primeira ideia é criar uma classe `CalculadoraDeSalario`, que recebe um `Funcionario` e retorna o salário do funcionário com o desconto já subtraído.

O primeiro cenário a ser testado será o de desenvolvedores com salários menor do que R$3000,0. Sabemos que o desconto é de 10%. Portanto, se o desenvolvedor ganhar R$1500,00, seu salário menos desconto deve ser de R$ 1350,00 (1500 * 90%):

```
public class CalculadoraDeSalarioTest {

    @Test
    public void
    deveCalcularSalarioParaDesenvolvedoresComSalarioAbaixoDoLimite() {

        CalculadoraDeSalario calculadora = new CalculadoraDeSalario();
        Funcionario desenvolvedor = new Funcionario
            ("Mauricio", 1500.0, Cargo.DESENVOLVEDOR);

        double salario = calculadora.calculaSalario(desenvolvedor);

        assertEquals(1500.0 * 0.9, salario, 0.00001);
    }
}
```

Hora de fazer o teste passar. Mas lembre-se que a ideia é fazer o teste passar da maneira **mais simples possível**. Veja o código abaixo:

```
public class CalculadoraDeSalario {

    public double calculaSalario(Funcionario funcionario) {
        return 1350.0;
    }
}
```

É possível ser mais simples do que isso? O código retorna diretamente o valor esperado pelo teste! Isso é aceitável, pois o código ainda não está finalizado.

Vamos agora ao próximo cenário: desenvolvedores que ganham mais do que R$ 3000.0. O teste é bem similar ao anterior:

```
@Test
public void
deveCalcularSalarioParaDesenvolvedoresComSalarioAcimaDoLimite() {

    CalculadoraDeSalario calculadora = new CalculadoraDeSalario();
    Funcionario desenvolvedor = new Funcionario
        ("Mauricio", 4000.0, Cargo.DESENVOLVEDOR);

    double salario = calculadora.calculaSalario(desenvolvedor);

    assertEquals(4000.0 * 0.8, salario, 0.00001);
}
```

Novamente, fazemos o teste passar da maneira mais simples possível. Veja o código abaixo:

```
public double calculaSalario(Funcionario funcionario) {
    if(funcionario.getSalario() > 3000) return 3200.0;
    return 1350.0;
}
```

Um simples *if* que verifica o salário do funcionário e retorna os valores esperados.

O próximo teste agora garante que DBAs com salários inferior a R$1500,00 recebem 15% de desconto:

```
@Test
public void deveCalcularSalarioParaDBAsComSalarioAbaixoDoLimite() {

    CalculadoraDeSalario calculadora = new CalculadoraDeSalario();
    Funcionario dba = new Funcionario
        ("Mauricio", 500.0, Cargo.DBA);

    double salario = calculadora.calculaSalario(dba);

    assertEquals(500.0 * 0.85, salario, 0.00001);
}
```

Para fazer esse teste passar da maneira mais simples, basta novamente colocar um outro *if*:

```
public double calculaSalario(Funcionario funcionario) {
    if(funcionario.getCargo().equals(Cargo.DESENVOLVEDOR)) {
        if(funcionario.getSalario() > 3000) return 3200.0;
        return 1350.0;
    }
    return 425.0;
}
```

Repare que, se for desenvolvedor, continuamos usando aquele código simples que escrevemos. Caso contrário, retornamos 425.0, que o valor esperado para o salário do DBA.

Mas agora, antes de continuarmos, precisamos discutir sobre o processo utilizado até agora.

4.3 Passos de Bebê (ou Baby Steps)

A ideia de sempre tomar o passo mais simples que resolva o problema naquele momento (e faça o teste passar) é conhecido pelos praticantes de TDD como **baby steps**. A vantagem desses passos de bebê é tentar levar o desenvolvedor sempre ao código mais simples e, por consequência, mais fácil de ser compreendido e mantido posteriormente.

O problema é que a ideia dos passos de bebê é muito mal interpretada por muitos praticantes. Por exemplo, veja que ao final, tínhamos 3 testes, 2 para as regras do desenvolvedor e 1 para a regra do DBA, e a implementação final gerada até então era:

```
public double calculaSalario(Funcionario funcionario) {
    if(funcionario.getCargo().equals(Cargo.DESENVOLVEDOR)) {
        if(funcionario.getSalario() > 3000) return 3200.0;
        return 1350.0;
    }
    return 425.0;
}
```

Repare que, a cada teste, nós implementamos a **modificação mais simples** (TDD prega por simplicidade), que era justamente adicionar mais um *if* e colocar o valor fixo que fazia o teste passar. Não há modificação mais simples do que essa.

O problema é que em algum momento fizemos tanto isso que nossa implementação passou a ficar muito longe da solução ideal. Veja que, dada a implementação

atual, levá-la para a solução flexível que resolverá o problema de forma correta não será fácil. O código ficou complexo sem percebermos.

Generalizando o ponto levantado: o desenvolvedor, ao invés de partir para uma solução mais abstrata, que resolveria o problema de forma genérica, prefere ficar postergando a implementação final, com a desculpa de estar praticando passos de bebê.

Imagine isso em um problema do mundo real, complexo e cheio de casos excepcionais. Na prática, o que acontece é que o programador gasta tanto tempo "contornando" a solução que resolveria o problema de uma vez que, quando chega a hora de generalizar o problema, ele se perde. Em sessões de Coding Dojo, onde os programadores se juntam para praticar TDD, é comum ver sessões que duram muito tempo e, ao final, os desenvolvedores pouco avançaram no problema.

Isso é simplesmente uma má interpretação do que são passos de bebê. O desenvolvedor deve buscar pela **solução mais simples, e não pela modificação mais simples**. Veja que a modificação mais simples não é necessariamente a solução mais simples.

No começo, modificações mais simples podem ajudar o desenvolvedor a pensar melhor no problema que está resolvendo; esse foi o caso quando implementamos o *return 1350.0*. Mas, caso o desenvolvedor continue somente implementando as modificações mais simples, isso pode afastá-lo da solução mais simples, que é o que o desenvolvedor deveria estar buscando ao final. Isso pode ser exemplificado pela sequência de *if* que fizemos onde, ao invés de partirmos para a solução correta (igual fizemos no capítulo anterior), optamos pela modificação mais simples.

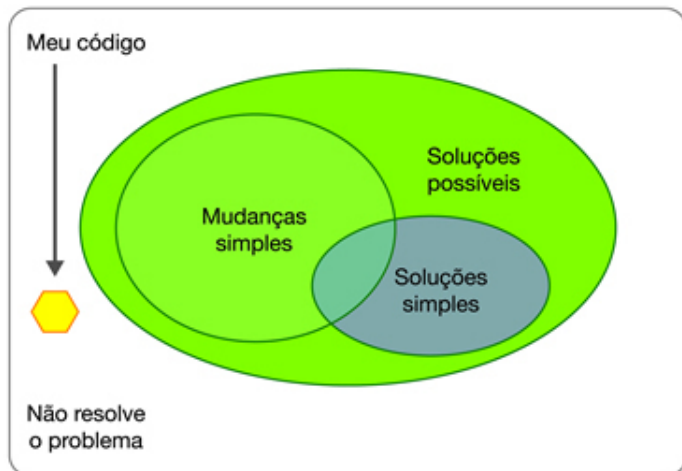
Coding Dojo

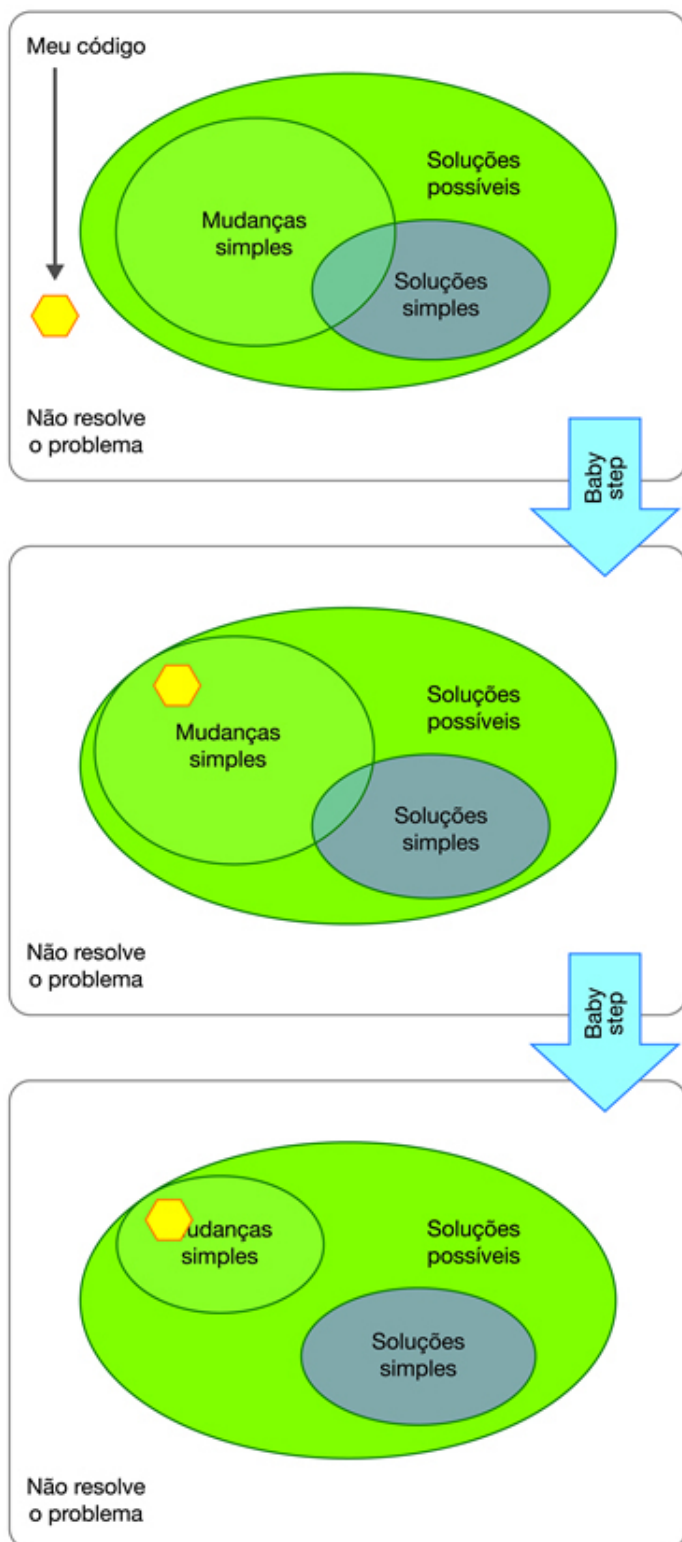
Um *coding dojo* é um encontro de desenvolvedores que estão interessados em aprender alguma coisa nova, como uma linguagem de programação diferente, ou técnicas de desenvolvimento diferentes. O objetivo é criar um ambiente tranquilo e favorável a novos aprendizados.

A prática mais comum é juntar alguns desenvolvedores e projetar uma máquina na parede. Dois desenvolvedores sentam em frente ao computador e começam a resolver algum problema pré-determinado, usando a linguagem e/ou a prática que desejam aprender melhor.

Geralmente esses dois desenvolvedores tem apenas alguns minutos para trabalhar. Enquanto eles trabalham a plateia assiste e eventualmente sugere alterações para o "piloto e co-piloto". Ao final do tempo (geralmente 7 minutos), o co-piloto vira piloto, o piloto volta para a plateia, e alguém da plateia se torna "co-piloto".

Você pode ler mais sobre coding dojos, diferentes estilos, problemas sugeridos nas mais diversas referências na Internet [7].

Veja as imagens abaixo. Nela temos o código e as possíveis soluções para o problema. Dentro desse conjunto, existem as que são resolvidas pelas mudanças mais simples, que podem ou não, ser a solução mais simples também para o problema:

Se o desenvolvedor continuamente opta pela modificação mais simples que não é a solução mais simples, isso só vai afastá-lo da melhor solução:

Meu código
Soluções possíveis
Mudanças simples
Soluções simples
Não resolve o problema
Baby step
Soluções possíveis
Mudanças simples
Soluções simples
Não resolve o problema
Baby step
udanças simples
Soluções possíveis
Soluções simples
Não resolve o problema

Em nosso exemplo, veja que a solução mais simples acabou sendo a utilização de *if*s. Sabemos que todo condicional faz o código ser mais difícil de ser mantido e de ser testado. Sabemos que essa não é a melhor solução para o problema; o uso de polimorfismo seria mais adequado. Nesse problema específico, discutiremos como resolvê-lo da melhor forma nos capítulos posteriores.

Novamente, resumindo a discussão: **A mudança mais simples para resolver um problema não é necessariamente a solução mais simples para resolvê-lo.**

Os passos de bebê servem para ajudar o desenvolvedor a entender melhor o problema e diminuir o passo caso ele não se sinta confortável com o problema que está resolvendo. Se o desenvolvedor está implementando um trecho de código complexo e têm dúvidas sobre como fazê-lo, ele pode desacelerar e fazer implementações mais simples; mas caso o desenvolvedor esteja seguro sobre o problema, ele pode tomar passos maiores e ir direto para uma implementação mais abstrata.

O próprio Kent Beck comenta sobre isso em seu livro. Em seu exemplo, ele implementa uma classe *Dinheiro*, responsável por representar uma unidade monetária e realizar operações sobre ela. No começo, ele toma passos simplórios, como os feitos acima, mas finaliza o capítulo dizendo que **ele não toma passos pequenos o tempo todo; mas que fica feliz por poder tomá-los quando necessário**.

Código duplicado entre o código de teste e o código de produção

Todo desenvolvedor sabe que deve refatorar qualquer duplicidade de código no momento que a encontrar. No próprio problema da soma, a duplicidade já aparece logo no primeiro teste: repare que o "1", "2" existentes no código de testes está duplicado com o "I" e "II" no código de produção. É fácil ver que cada "I", adiciona 1 no número final. Sim, você deve ficar atento com duplicidade de código entre a classe de teste e a classe de produção. Nesse caso, após o teste ficar verde, o desenvolvedor poderia já refatorar o código de produção para algo mais genérico, e resolver a duplicidade.

O mesmo acontece em nosso código. No momento em que colocamos o valor "1350.0" fixo em nosso código de produção, geramos uma duplicação. O mesmo valor estava no código de teste: "1500.0 * 0.9". Nesse momento, o desenvolvedor já poderia ter refatorado e generalizado a solução para algo como `funcionario.getSalario() * 0.9`.

4.4 Usando baby steps de maneira consciente

De volta ao problema da calculadora de salário, vamos repetir a implementação, mas dessa vez usando passos de bebê de forma consciente e produtiva.

Novamente, o primeiro teste:

```
@Test
public void
deveCalcularSalarioParaDesenvolvedoresComSalarioAbaixoDoLimite() {

    CalculadoraDeSalario calculadora = new CalculadoraDeSalario();
    Funcionario desenvolvedor = new Funcionario
        ("Mauricio", 1500.0, Cargo.DESENVOLVEDOR);

    double salario = calculadora.calculaSalario(desenvolvedor);

    assertEquals(1500.0 * 0.9, salario, 0.00001);
}
```

Vamos fazê-lo passar da maneira mais simples possível:

```
public double calculaSalario(Funcionario funcionario) {
    return 1500 * 0.9;
}
```

Mas, dessa vez, ao contrário de partir para o próximo teste, vamos remover a duplicação de dado que existe entre o código de teste e o código de produção:

```
public double calculaSalario(Funcionario funcionario) {
    return funcionario.getSalario() * 0.9;
}
```

Já está melhor. O código já está genérico e mais perto da solução final. O próximo teste garante o desconto correto para desenvolvedores que ganham acima do limite:

```
@Test
public void
deveCalcularSalarioParaDesenvolvedoresComSalarioAcimaDoLimite() {

    CalculadoraDeSalario calculadora = new CalculadoraDeSalario();
    Funcionario desenvolvedor = new Funcionario
        ("Mauricio", 4000.0, Cargo.DESENVOLVEDOR);
```

```
    double salario = calculadora.calculaSalario(desenvolvedor);

    assertEquals(4000.0 * 0.8, salario, 0.00001);
}
```

Nesse momento, não há dúvidas de como será a implementação. Precisamos verificar o salário do desenvolvedor. Se ele for maior que R$3000,00, devemos descontar 20%. Caso contrário, descontamos os mesmos 10%. Como a implementação está bem clara em nossa cabeça, podemos ir direto para a solução final:

```
public double calculaSalario(Funcionario funcionario) {
    if(funcionario.getSalario() > 3000) {
        return funcionario.getSalario() * 0.8;
    }
    return funcionario.getSalario() * 0.9;
}
```

Veja a diferença dos passos de bebê nos dois testes que escrevemos. No primeiro, tomamos a decisão mais simples possível na hora de fazer o teste passar. Mas logo em seguida, seguimos o ciclo de TDD à risca: removemos a duplicação de código que já existia. No segundo teste, estávamos seguros de como fazer a implementação, e a buscamos diretamente.

Será que praticamos TDD errado? Lembre-se que o objetivo do desenvolvedor não é praticar TDD, mas sim entregar código de qualidade. TDD e passos de bebê estão lá para ajudar, mas não são regra de ouro. Um bom programador sabe a hora de ir mais devagar (geralmente quando está passando por um problema complicado do ponto de vista lógico ou do ponto de vista de design de classes) e realmente usar passos de bebê, e também sabe a hora de acelerar um pouco mais (geralmente quando não há muitas dúvidas sobre como resolver o problema ou sobre como projetar a classe), ganhando em produtividade.

Continuaremos a implementação desta calculadora nos próximos capítulos. Vamos agora finalizar nossa discussão sobre passos de bebê.

Uncle Bob e os passos de refatoração do TDD

Há alguns anos atrás, Robert Martin fez um post em seu blog sobre uma suposta sequência de passos de refatoração que levariam o desenvolvedor ao melhor código [25]. Seguindo a ideia dos baby steps, ele comenta que um código de produção começa bem específico e caminha até em direção a generalização, para que atenda o problema por completo.

Segundo ele, na hora de generalizar o código, se você usasse baby steps e seguisse as refatorações na sequência que ele descreveu, você sempre chegaria ao melhor algoritmo possível para aquele código. Era como se ele tivesse uma maneira mecânica para se produzir código de qualidade.

Alguns dias depois, eu e o Guilherme Silveira respondemos ao post, mostrando nosso ponto contrário. Em nossa opinião, não é possível criar uma simples sequência de passos para se chegar ao melhor algoritmo possível. Novamente, a experiência e conhecimento do desenvolvedor são necessários.

Por fim, o ponto é que é realmente difícil saber o tamanho do passo a ser dado, e como refatorar o código para que ele fique cada vez melhor. Sua experiência deve ser levada em conta nessa hora.

Vantagem de fazer o teste passar rápido

Uma das vantagens de fazer o teste passar de maneira simples e rápida é "testar o teste". Seu teste é código; e ele pode ter bugs também.

Como não faz sentido escrever um teste para o teste, uma maneira de testá-lo é garantir que ele falhe quando precisa falhar, e passe quando precisa passar. Ou seja, antes de começar a implementação, veja o teste falhar. Com ele falhando, tente fazê-lo passar da maneira mais simples possível, com o objetivo apenas de vê-lo ficando verde. Dessa forma, você "testou o teste".

4.5 Conclusão

Neste capítulo, discutimos sobre os famosos **baby steps**. A ideia é possibilitar ao desenvolvedor andar na velocidade que achar necessário. Se o código que ele está implementando naquele momento é complicado e/ou complexo, tomar passos de bebê podem ajudá-lo a entender melhor o problema e a buscar por soluções mais simples.

Mas, se o código que ele está trabalhando é algo que já está bem claro e não há muitas dúvidas sobre a implementação, o desenvolvedor pode então dar um passo um pouco maior, já que passos pequenos farão com que ele apenas diminua sua produtividade.

No fim, a grande vantagem dos passos de bebê é poder aprender algo sobre o código para que o mesmo possa ser melhorado. Se o desenvolvedor não está aprendendo nada, então talvez não haja razão para dar passos de bebê naquele instante.

Discutimos também que **a modificação mais simples não é necessariamente a solução mais simples que resolve o problema**. Desenvolvedores devem buscar simplicidade não apenas no nível de código, mas também no nível de design de classes. Às vezes um simples *if* pode ser o código mais simples a ser escrito, mas talvez não seja a melhor solução do ponto de vista do design.

Use baby steps com parcimônia. Um bom praticante de TDD sabe a hora de aumentar ou diminuir o passo. Use os passos de bebê para o bem do seu projeto, e não simplesmente porque é uma regra. Parafraseando Jason Gorman, *"se fazer a coisa mais simples significa fazer uso de muitos ifs ou switchs, muito provavelmente você não entendeu TDD.*"

Capítulo 5

TDD e Design de Classes

Como dito anteriormente, TDD é bastante popular pelos seus efeitos positivos no design das classes do nosso sistema. Neste capítulo, começaremos a discutir como os testes podem efetivamente ajudar desenvolvedores a pensar melhor em relação às classes que estão criando.

5.1 O Problema do Carrinho de Compras

Suponha agora que o projeto atual seja uma loja virtual. Essa loja virtual possui um carrinho de compras, que guarda uma lista de itens comprados. Um item possui a descrição de um produto, a quantidade, o valor unitário e o valor total desse item. Veja o código que representa esse carrinho:

```
public class CarrinhoDeCompras {

    private List<Item> itens;
```

```
    public CarrinhoDeCompras() {
        this.itens = new ArrayList<Item>();
    }

    public void adiciona(Item item) {
        this.itens.add(item);
    }

    public List<Item> getItens() {
        return Collections.unmodifiableList(itens);
    }
}
```

Um item também é uma simples classe:

```
public class Item {

    private String descricao;
    private int quantidade;
    private double valorUnitario;

    public Item(String descricao,
            int quantidade,
            double valorUnitario) {
        this.descricao = descricao;
        this.quantidade = quantidade;
        this.valorUnitario = valorUnitario;
    }

    public double getValorTotal() {
        return this.valorUnitario * this.quantidade;
    }

    // getters para os atributos

}
```

Agora imagine que o programador deva implementar uma funcionalidade que devolva o valor do item de maior valor dentro desse carrinho de compras. Pensando já nos testes, temos os seguintes cenários:

- Se o carrinho só tiver um item, ele mesmo será o item de maior valor.

- Se o carrinho tiver muitos itens, o item de maior valor é o que deve ser retornado.
- Um carrinho sem nenhum item deve retornar zero.

Seguindo TDD a risca, vamos começar pelo cenário mais simples, que nesse caso é o carrinho vazio. Vamos criar um teste para a classe *MaiorPreco*, responsável por essa tarefa:

```
public class MaiorPrecoTest {

    @Test
    public void deveRetornarZeroSeCarrinhoVazio() {
        CarrinhoDeCompras carrinho = new CarrinhoDeCompras();

        MaiorPreco algoritmo = new MaiorPreco();
        double valor = algoritmo.encontra(carrinho);

        assertEquals(0.0, valor, 0.0001);
    }
}
```

Fazer esse teste passar é fácil; basta retornar zero.

```
public class MaiorPreco {

    public double encontra(CarrinhoDeCompras carrinho) {
        return 0;
    }

}
```

Ainda não há uma repetição de código grande a ser eliminada, então podemos ir para o próximo teste, que é o caso do carrinho conter apenas um produto. O teste:

```
@Test
public void deveRetornarValorDoItemSeCarrinhoCom1Elemento() {
    CarrinhoDeCompras carrinho = new CarrinhoDeCompras();
    carrinho.adiciona(new Item("Geladeira", 1, 900.0));

    MaiorPreco algoritmo = new MaiorPreco();
    double valor = algoritmo.encontra(carrinho);
```

```
    assertEquals(900.0, valor, 0.0001);

}
```

Para fazer esse teste passar, precisaremos escrever um pouco mais de código. Mas não há muitas dúvidas sobre o algoritmo nesse momento: se o carrinho estiver vazio, retorna 0. Caso contrário, retorna o valor do primeiro elemento desse carrinho:

```
public double encontra(CarrinhoDeCompras carrinho) {
    if(carrinho.getItens().size() == 0) return 0;
    return carrinho.getItens().get(0).getValorTotal();
}
```

Por fim, o cenário que resta: é necessário encontrar o item de maior valor caso o carrinho contenha muitos itens:

```
@Test
public void deveRetornarMaiorValorSeCarrinhoContemMuitosElementos() {
    CarrinhoDeCompras carrinho = new CarrinhoDeCompras();
    carrinho.adiciona(new Item("Geladeira", 1, 900.0));
    carrinho.adiciona(new Item("Fogão", 1, 1500.0));
    carrinho.adiciona(new Item("Máquina de Lavar", 1, 750.0));

    MaiorPreco algoritmo = new MaiorPreco();
    double valor = algoritmo.encontra(carrinho);

    assertEquals(1500.0, valor, 0.0001);

}
```

Vamos à implementação. É necessário navegar pelos itens da coleção, procurando pelo item de maior valor total. Para armazenar esse maior valor é necessário um atributo da classe, batizado de “maior":

```
public double encontra(CarrinhoDeCompras carrinho) {
    if(carrinho.getItens().size() == 0) return 0;

    double maior = carrinho.getItens().get(0).getValorTotal();
    for(Item item : carrinho.getItens()) {
        if(maior < item.getValorTotal()) {
            maior = item.getValorTotal();
```

```
        }
    }

    return maior;
}
```

Com todos os testes passando, é hora de avaliá-los. Observe, por exemplo, o último teste que escrevemos. Veja que o teste instancia a classe `MaiorPreco`, passa para ela um "carrinho" e verifica o retorno desse método. Repare que todo o cenário montado foi em cima da classe `CarrinhoDeCompras`; não fizemos nada na classe `MaiorPreco`.

Isso pode ser um mau sinal. Sabemos que uma das vantagens da orientação a objetos em relação a códigos mais procedurais é justamente a capacidade de unir dados e comportamentos. A classe `MaiorPreco` parece "estranha", já que não houve necessidade de setar atributos ou qualquer outro dado nela.

Sabendo desse possível problema na classe, vamos ver seu código. Repare que o método `encontra()` faz uso do `carrinho` praticamente o tempo todo, e não faz uso de nada específico da sua própria classe.

A pergunta é: por que criamos essa classe? Por que o método `encontra()` não está dentro da própria classe `CarrinhoDeCompras` ?

Vamos fazer essa mudança. Se levarmos toda a lógica do método `encontra()` para um método chamado, por exemplo, `maiorValor()` dentro do `CarrinhoDeCompras`, a classe ficará assim:

```
public double maiorValor() {
    if(itens.size() == 0) return 0;

    double maior = itens.get(0).getValorTotal();
    for(Item item : itens) {
        if(maior < item.getValorTotal()) {
            maior = item.getValorTotal();
        }
    }

    return maior;
}
```

Já os testes ficarão assim:

```java
public class CarrinhoDeComprasTest {

    @Test
    public void deveRetornarZeroSeCarrinhoVazio() {
        CarrinhoDeCompras carrinho = new CarrinhoDeCompras();

        assertEquals(0.0, carrinho.maiorValor(), 0.0001);
    }

    @Test
    public void deveRetornarValorDoItemSeCarrinhoCom1Elemento() {
        CarrinhoDeCompras carrinho = new CarrinhoDeCompras();
        carrinho.adiciona(new Item("Geladeira", 1, 900.0));

        assertEquals(900.0, carrinho.maiorValor(), 0.0001);
    }

    @Test
    public void deveRetornarMaiorValorSeCarrinhoContemMuitosElementos() {
        CarrinhoDeCompras carrinho = new CarrinhoDeCompras();
        carrinho.adiciona(new Item("Geladeira", 1, 900.0));
        carrinho.adiciona(new Item("Fogão", 1, 1500.0));
        carrinho.adiciona(new Item("Máquina de Lavar", 1, 750.0));

        assertEquals(1500.0, carrinho.maiorValor(), 0.0001);
    }
}
```

Veja agora que nosso teste está muito mais claro e elegante. Os testes do `CarrinhoDeCompras` instanciam um carrinho, montam o cenário desejado e invocam um método do próprio carrinho. Agora o código dentro do teste parece melhor e mais orientado a objetos. O comportamento está no lugar esperado. É justamente isso que se espera de um código de qualidade.

Antes de continuarmos, vamos refletir um pouco sobre como descartamos a primeira solução e chegamos à solução que melhor resolveu o problema.

5.2 Testes que influenciam no design de classes

O que nos fez optar pelo segundo caminho ao invés do primeiro? Em que momento mudamos o rumo da implementação? Releia o texto acima. Repare que o que nos

fez mudar de ideia foi justamente o fato de termos encontrado um teste com uma característica estranha (a não-necessidade de um cenário para a classe sob teste).

Esse é o principal ponto deste capítulo. Muitos praticantes de TDD afirmam que a prática lhes guia no projeto de classes. A grande pergunta é: como isso acontece? Isso é outro ponto bastante curioso e mal entendido sobre TDD. Apesar de muitos praticantes de TDD acreditarem que a prática guia a criação do design, poucos sabem explicar como isso realmente acontece. Esta é, aliás, o grande ponto que este livro tenta atacar.

A prática de TDD pode influenciar no processo de criação do projeto de classes. No entanto, ao contrário do que afirmações mais superficiais fazem parecer, **a prática de TDD não guia o desenvolvedor para um bom projeto de classes de forma automática; a experiência e conhecimento do desenvolvedor são fundamentais ao criar software orientado a objetos**.

A prática, por meio dos seus possíveis *feedback* em relação ao projeto de classes, que serão discutidos em detalhes nos capítulos a seguir, pode servir de guia para o desenvolvedor. Esses *feedback*, quando observados, fazem com que o desenvolvedor perceba problemas de projeto de classes de forma antecipada, facilitando a refatoração do código. Um exemplo desses *feedback* foi justamente o que observamos no primeiro teste que escrevemos. O teste não montava qualquer cenário na classe que estava sob teste. Isso simplesmente não parecia correto. São pequenas dicas como essas, que ajudam o desenvolvedor a optar por um melhor projeto de classes.

Portanto, esta é a forma na qual a prática guia o desenvolvedor para um melhor projeto de classes: dando retorno constante sobre os possíveis problemas existentes no atual projeto de classes. É tarefa do desenvolvedor perceber esses problemas e melhorar o projeto de acordo.

5.3 Diferenças entre TDD e testes da maneira tradicional

O desenvolvedor que pratica TDD escreve os testes antes do código. Isso faz com que o teste de unidade que está sendo escrito sirva de rascunho para o desenvolvedor. Foi novamente o que fizemos. Começamos pelo teste e, ao perceber que o teste "estava estranho", o jogamos fora e começamos outro. O custo de jogá-lo fora naquele momento era quase zero. Não havíamos escrito uma linha na classe de produção. Tínhamos a chance de começar de novo, sem qualquer perda.

Mas o mesmo *feedback* poderia ser obtido caso o desenvolvedor deixasse para es-

crever o teste ao final da implementação. Mas então qual seria a vantagem da prática de TDD?

Ao deixar a escrita do teste somente para o final, no momento que o desenvolvedor perceber um problema de design graças ao teste, o custo de mudança será alto. Leve isso para o mundo real, onde o programador escreve dezenas de classes que interagem entre si e só depois ele escreve o teste.

Todas as decisões de design (boas ou ruins) já foram tomadas. Mudar algo nesse momento pode ser caro, afinal uma mudança em uma classe pode impactar em muitas mudanças nas outras classes que dependem da classe alterada.

O praticante de TDD não tem esse problema. Afinal, graças a escrita do teste antes e à simplicidade inerente do processo, ele escreve um teste para uma pequena funcionalidade, recebe *feedback* dela, faz as mudanças que achar necessário e implementa. Depois, ele repete o ciclo. Ou seja, o praticante de TDD recebe *feedback* constante ao longo do seu dia a dia de trabalho, fazendo-o encontrar possíveis problemas de design mais cedo e, por consequência, diminuindo o custo de mudança.

Fishbowling no Encontro Ágil

Ao longo do meu mestrado, participei e conversei com diversas pessoas praticantes ou interessadas em TDD. Em um desses eventos, o Encontro Ágil de 2010, organizei um pequeno debate sobre o assunto [27]. Éramos em torno de 12 pessoas discutindo sobre TDD (eu não participei da discussão, apenas a moderei), e ficamos lá por 1h30.

E durante toda a conversa, praticamente todos os participantes afirmavam firmemente que TDD os ajudava a produzir classes melhores. Mas, quando perguntei "como isso acontece exatamente", todos eles passaram a falar menos. Eles não sabiam dizer como a prática os guiava, apenas "sentiam" que isso acontecia.

Essa foi uma das grandes motivações da minha pesquisa de mestrado e deste livro sobre o assunto. Realmente não é claro para todos como TDD guia o desenvolvedor. Espero com este livro mostrar, de maneira explícita, como isso acontece.

5.4 Testes como rascunho

Observe o teste abaixo:

```
@Test
public void deveRetornarValorDoItemSeCarrinhoCom1Elemento() {
    CarrinhoDeCompras carrinho = new CarrinhoDeCompras();
    carrinho.adiciona(new Item("Geladeira", 1, 900.0));

    assertEquals(900.0, carrinho.maiorValor(), 0.0001);
}
```

Repare que, apesar de simples, ele deixa bem claro uma grande quantidade de decisões tomadas na classe `CarrinhoDeCompras`:

- O nome do método que calculará o maior valor (`maiorValor()`)
- O nome do método que recebe um novo item (`adiciona()`)
- Os parâmetros que esses métodos recebem (um `Item`)
- O retorno do método (um *double*)

Todas elas são decisões de design! Se alguma dessas decisões não agradar o programador, ele pode simplesmente mudar de ideia. O teste serve como um rascunho para o desenvolvedor, onde ele pode experimentar as diferentes maneiras de se projetar a classe.

E o teste, na verdade, é um excelente rascunho. O teste é o primeiro "cliente" da classe que está sendo criada. Ele é a primeira classe que está interagindo com ela. E olhar para o teste com esse ponto de vista pode fazer sentido. Se está difícil testar um comportamento, é porque, muito provavelmente, sua classe está mal projetada.

Um teste no fim das contas é apenas uma classe que instancia a classe sob teste e invoca um comportamento (método), passando um cenário determinado. Como isso poderia ser difícil? Se está, novamente, talvez sua classe esteja muito acoplada ou seja pouco coesa (possua responsabilidades demais). Um programador atento percebe que está difícil escrever o teste e muda de ideia em relação ao design da classe que está criando. Testar deve ser uma atividade fácil e prazerosa.

O próprio Michael Feathers [18] já escreveu um post sobre isso, e ele sumariza a discussão dizendo que **existe uma relação muito forte entre uma classe fácil de testar e uma classe que está bem projetada**.

Dado essa grande relação entre um código fácil de ser testado e um código de qualidade, será difícil separar TDD de design de classes daqui para a frente. Por esse

motivo, ao final do livro, encontra-se um apêndice sobre design de classes. Nele, discutimos os **princípios SOLID**, um conjunto de 5 boas ideias de orientação a objetos. Esses princípios discutem basicamente as vantagens de se ter classes coesas, pouco acopladas e simples. Caso o leitor não esteja familiarizado com estes princípios, é sugerido que ele o leia; conhecer estes princípios facilitará a leitura dos próximos capítulos.

5.5 Conclusão

Já vimos que os testes podem influenciar o design das classes que estamos desenvolvendo. Ao observar o código do teste de unidade com atenção, o desenvolvedor pode perceber problemas no projeto de classes que está criando. Problemas esses como classes que possuem diversas responsabilidades ou que possuem muitas dependências.

Ao escrever um teste de unidade para uma determinada classe, o desenvolvedor é obrigado a passar sempre pelos mesmos passos: a escrita do cenário, a execução da ação sob teste e, por fim, a garantia que o comportamento foi executado de acordo com o esperado. Uma dificuldade na escrita de qualquer uma dessas partes pode implicar em problemas no projeto de classes. O desenvolvedor, atento, percebe e melhora seu projeto de classes de acordo.

Ao longo do livro, descreveremos padrões que podem ajudar o praticante a perceber problemas de design específicos, como baixa coesão, alto acoplamento e complexidade desnecessária.

Capítulo 6

Qualidade no Código do Teste

No capítulo anterior, discutimos que os testes podem eventualmente indicar problemas no código do teste. Com essa informação em mãos, desenvolvedores podem refatorar o código e melhorá-lo. Mas para que isso aconteça, o código do teste deve ser o mais claro e limpo possível.

Escrever testes deve ser algo fácil e produtivo. Todas as boas práticas que o desenvolvedor aplica no código de produção pode ser utilizado no código do teste também, para que ele fique mais claro e mais reutilizável.

Nas seções a seguir, discutiremos práticas que tornam os testes mais fáceis de ler e manter.

A equipe que odiava os testes

Uma vez conversando com um aluno, ele me comentou que a equipe escrevia testes e que o projeto que atuavam possuía uma bateria de testes imensa, mas pasmem: eles odiavam os testes!

Curioso pelo assunto, comecei a fazer perguntas para entender melhor qual era o real problema deles. Após alguns minutos de conversa, descobri que o que os incomodava era o fato de que qualquer alteração simples no código de produção impactava em profundas alterações no código dos testes. Ou seja, eles gastavam 30 segundos implementando uma nova funcionalidade, e 30 minutos alterando testes.

Por que isso aconteceu? A resposta é simples: falta de cuidado com o código dos testes. Agora nossos testes são automatizados, eles são código. E código ruim impacta profundamente na produtividade da equipe.

Todo o carinho que os desenvolvedores tem com seus códigos de produção deve acontecer também com o código de testes. Evitar repetição de código, isolar responsabilidades em classes específicas e etc, são fundamentais para que sua bateria de testes viva para sempre.

6.1 Repetição de código entre testes

Veja a classe *CarrinhoDeComprasTest*. Ela contém apenas 3 testes, mas repare na quantidade de código repetido que já existe. Veja que todos os testes possuem a linha que instancia o carrinho de compras. E, muito provavelmente, os próximos testes para essa classe conterão o mesmo conjunto de linhas.

```
public class CarrinhoDeComprasTest {

    @Test
    public void deveRetornarZeroSeCarrinhoVazio() {
        CarrinhoDeCompras carrinho = new CarrinhoDeCompras();

        // continuação do teste aqui...
    }

    @Test
    public void deveRetornarValorDoItemSeCarrinhoCom1Elemento() {
```

```
        CarrinhoDeCompras carrinho = new CarrinhoDeCompras();

        // continuação do teste aqui...
    }

    @Test
    public void deveRetornarMaiorValorSeCarrinhoContemMuitosElementos(){
        CarrinhoDeCompras carrinho = new CarrinhoDeCompras();

        // continuação do teste aqui...
    }
}
```

Imagine se, por algum motivo, o construtor da classe `CarrinhoDeCompras` mudar. A mudança deverá acontecer em quantos testes? Imagine se essa classe contivesse 100 testes; seriam necessários alterar em 100 pontos diferentes.

É bem claro para todo desenvolvedor que qualquer código repetido é problemático. Se surge a necessidade de mudar esse código, ele deve alterar em todos os pontos cujo código se repete. Isso torna a manutenção cara. O mesmo ponto vale para os testes.

A solução, portanto, é escrever esse código em apenas um único lugar. Algo como um método de inicialização da classe de teste. O JUnit possui uma alternativa simples e elegante para isso. Basta anotar o método com *@Before* e o JUnit se encarregará de invocar esse método antes de cada teste. Veja como o código fica muito mais claro, com o método *inicializa* centralizando a criação do cenário inicial. Cada método de teste agora é muito mais simples de ser lido:

```
public class CarrinhoDeComprasTest {

    private CarrinhoDeCompras carrinho;

    @Before
    public void inicializa() {
        this.carrinho = new CarrinhoDeCompras();
    }

    @Test
    public void deveRetornarZeroSeCarrinhoVazio() {
        assertEquals(0.0, carrinho.maiorValor(), 0.0001);
    }
```

```
    @Test
    public void deveRetornarValorDoItemSeCarrinhoCom1Elemento() {
        carrinho.adiciona(new Item("Geladeira", 1, 900.0));

        assertEquals(900.0, carrinho.maiorValor(), 0.0001);

    }

    @Test
    public void deveRetornarMaiorValorSeCarrinhoContemMuitosElementos(){
        carrinho.adiciona(new Item("Geladeira", 1, 900.0));
        carrinho.adiciona(new Item("Fogão", 1, 1500.0));
        carrinho.adiciona(new Item("Máquina de Lavar", 1, 750.0));

        assertEquals(1500.0, carrinho.maiorValor(), 0.0001);
    }
}
```

Fazer uso de métodos de inicialização é sempre uma boa ideia. Todo e qualquer código repetido nos testes pode ser levado para métodos como esse. A vantagem? Novamente, código claro.

Quando usar @After?

Análoga ao `@Before`, a anotação `@After` diz ao JUnit para executar esse método logo após a execução do teste.

Geralmente fazemos uso dele quando queremos "limpar" alguma sujeira que o teste criou. Em testes puramente de unidade, raramente os utilizamos. Mas agora imagine um teste que garante que um registro foi salvo no banco de dados, ou que um arquivo XML foi escrito no disco. Após executarmos o método de teste, precisaremos limpar a tabela ou deletar o arquivo gerado.

Portanto, apesar do `@After` não ser útil em testes de unidade, ele é extremamente importante em testes de integração.

6.2 Nomenclatura dos testes

Outro ponto igualmente importante em um teste de unidade é o seu nome. Escolher um bom nome pode fazer uma grande diferença na produtividade do desenvolvedor.

Veja por exemplo, os métodos de teste apresentados na seção anterior. Eles validam o funcionamento do método *maiorValor()*. Imagine que a classe de teste fosse parecido com a abaixo:

```
public class CarrinhoDeComprasTest {
    @Test public void maiorValor_1() { /* ... */ }
    @Test public void maiorValor_2() { /* ... */ }
    @Test public void maiorValor_3() { /* ... */ }
}
```

Se um desses testes falhar, o programador precisará ler o código do teste para entender qual o cenário que faz o comportamento quebrar.

Lembre-se que projetos contém inúmeros testes de unidade, e se o desenvolvedor precisar ler todo o corpo de um método de teste (mais seu método de inicialização) para entender o que ele faz, perderá muito tempo. É um fato que ler código é muito mais difícil do que escrever código [30]. A escolha de um bom nome do teste pode ajudar o desenvolvedor a ler menos.

Uma sugestão é deixar bem claro no nome do teste qual o comportamento esperado da classe perante determinado cenário. Por exemplo, veja os nomes de testes que demos anteriormente:

```
public class CarrinhoDeComprasTest {
    @Test public void deveRetornarZeroSeCarrinhoVazio()
    { /* ... */ }
    @Test public void deveRetornarValorDoItemSeCarrinhoCom1Elemento()
    { /* ... */ }
    @Test
    public void deveRetornarMaiorValorSeCarrinhoContemMuitosElementos()
    { /* ... */ }
}
```

Um método cujo nome tenha um tamanho como esse seria considerado um pecado em um método de produção. Mas veja que basta o desenvolvedor ler o nome do teste para entender quais são os comportamentos esperados desta classe.

Não há uma regra especial para nomear um método de teste (assim como não há nenhuma regra clara para nomear classes, métodos e variáveis) e, portanto, sua

equipe pode definir o padrão que achar melhor. A única ressalva, novamente, é que seu nome deixe claro qual o comportamento esperado e validado naquele método de teste.

Underscore no nome do teste

O Java utiliza camel case como convenção para nomes de métodos e atributos. Ou seja, sempre que desejamos escrever um método com N palavras, deixamos a primeira em minúscula, e todas as outras com a sua primeira letra em maiúsculo.

Mas como os nomes dos métodos de teste são extensos (`deveRetornarValorDoItemSeCarrinhoCom1Elemento`), muitos desenvolvedores optam por usar underscore (_) para separar as palavras e facilitar a leitura. Por exemplo, os testes da classe `CarrinhoDeCompras` ficariam assim:

```
public class CarrinhoDeComprasTest {
    @Test
    public void deve_retornar_zero_se_carrinho_vazio()
    { /* ... */ }
    @Test
    public void deve_retornar_valor_do_item_se_carrinho_com_1_elemento()
    { /* ... */ }
    @Test
    public void
    deve_retornar_maior_valor_se_carrinho_contem_muitos_elementos()
    { /* ... */ }
}
```

Se a equipe achar que isso facilitará a leitura e manutenção dos testes, faça. Caso contrário, fique na convenção da sua linguagem.

6.3 Test Data Builders

Uma outra maneira de reduzir a quantidade de linhas gastas com a criação do cenário é criar classes que auxiliam nesse trabalho. Por exemplo, veja a classe *Carrinho*. Para criar um, é necessário passar uma lista de itens. Ou seja, em todos os futuros testes que serão escritos, gastaremos ao menos 2 linhas para se criar um carrinho. Isso não parece uma boa ideia.

Veja o pseudocódigo abaixo. Com apenas 1 linha de código, criamos um *Carrinho* com 2 itens no valor de 200,00 e 300,00 reais, respectivamente, pronto para ser utilizado pelos testes.

```
public class CarrinhoDeComprasBuilder {

    public CarrinhoDeCompras cria() { /* ... */}
    public CarrinhoDeComprasBuilder comItens(...) { /* ... */}
}

CarrinhoDeCompras carrinho = new CarrinhoDeComprasBuilder()
    .comItens(200.0, 300.0)
    .cria();
```

Fazendo uso do padrão de projeto Builder [19], podemos facilmente criar uma classe como essa. Veja o código abaixo:

```
public class CarrinhoDeComprasBuilder {

    public CarrinhoDeCompras carrinho;

    public CarrinhoDeComprasBuilder() {
        this.carrinho = new CarrinhoDeCompras();
    }

    public CarrinhoDeComprasBuilder comItens(double... valores) {
        for(double valor : valores) {
            carrinho.adiciona(new Item("item", 1, valor));
        }
        return this;
    }

    public CarrinhoDeCompras cria() {
        return carrinho;
    }
}
```

A classe *CarrinhoDeComprasBuilder* possibilita a criação de um carrinho com itens de maneira fácil e rápida. Esse builder pode ainda ser melhorado, dando opções para se criar itens com diferentes quantidades e etc. Lembre-se que você tem todo o poder da orientação a objetos aqui para criar classes que facilitem sua vida.

No fim, você pode implementar Builders da maneira que achar mais conveniente. Assim como em todo padrão de projeto, os detalhes de implementação não são a parte mais importante. Lembre-se que a ideia é facilitar o processo de criação

de objetos que serão utilizados em cenários de teste. Classes como essas, recebem o nome de **Test Data Builders** [4].

Test Data Builders são especialmente úteis quando o teste envolve um cenário complexo, cheio de entidades com os mais diferentes valores. Imagine no mundo real, onde cenários são grandes e complicados. Não fazer uso de Builders implica em repetição de muitas linhas de código para se criar cenários semelhantes.

Uma outra vantagem da utilização de Test Data Builders ao longo do código de teste é a facilidade na evolução desse mesmo código. Se a classe `CarrinhoDeCompras` mudar, todos as classes de teste que fazem uso dela precisarão mudar também. Isso pode ser caro. Ao contrário, se o desenvolvedor sempre cria seus cenários a partir de Test Data Builders, qualquer mudança na classe será refletida apenas no Builder; bastará ao desenvolvedor mexer nele e todos os testes voltarão a funcionar.

6.4 Testes Repetidos

Uma frase comum entre desenvolvedores de software é que *"melhor do que escrever código, é apagar código!"*. A pergunta é: faz sentido apagar um teste?

A primeira e mais óbvia resposta é: apague o teste quando ele deixar de fazer sentido. Se a funcionalidade foi removida, o desenvolvedor deve atualizar a bateria de testes e apagar todos os testes relacionados à ela. Se a funcionalidade evoluir, você deve evoluir seus testes juntos. Uma bateria de testes desatualizada não serve de nada; só atrapalha o andamento da equipe.

A segunda resposta é: quando a bateria de testes conter testes repetidos. Em algumas situações, desenvolvedores ficam em dúvida sobre como implementar determinada funcionalidade, e optam por escrever testes que recebem entradas semelhantes. Isso serve para que consigam tomar passos menores (baby steps) e cheguem na solução mais simples para o problema.

No exemplo da Calculadora, o desenvolvedor criaria testes para a soma de 1+1, 1+2 e 2+2, por exemplo:

```
public class CalculadoraTest {
  @Test
  public void deveSomarUmMaisUm() {
    assertEquals(2, new Calculadora().soma(1,1));
  }
```

```
  @Test
  public void deveSomarUmMaisDois() {
    assertEquals(3, new Calculadora().soma(1,2));
  }

  @Test
  public void deveSomarDoisMaisDois() {
    assertEquals(4, new Calculadora().soma(2,2));
  }
}
```

Esses testes, muito úteis durante o tempo de desenvolvimento do algoritmo, agora se tornaram repetidos. O desenvolvedor deve, portanto, apagá-los. Eles, além de serem inúteis, ainda dificultam o trabalho do desenvolvedor. Se um dia o método testado mudar seu comportamento ou sua interface pública, o desenvolvedor terá que mudar em 10, 20 testes diferentes (mas que no fim testam a mesma coisa). Lembre-se do acoplamento entre seu código de teste e seu código de produção.

Mas será que apenas um teste por funcionalidade é suficiente? Sim. Uma bateria de testes deve ter apenas um teste de unidade para cada conjunto de estados válidos e inválidos para uma condição de entrada. Espera-se que todos os elementos de uma classe se comportem de maneira similar dado duas entradas semelhantes.

Esses conjuntos são conhecidos por **classes de equivalência** [23]. Escrever apenas um teste por classe de equivalência é uma prática muito comum em testes de caixa preta e é conhecida como particionamento em classes de equivalência. Essa prática também faz muito sentido em testes de caixa branca, como os testes de unidade.

No exemplo da calculadora, poderíamos ter testes para as seguintes classes de equivalência:

- soma de dois números positivos;
- soma de um número positivo com outro negativo;
- soma de um número negativo com outro positivo;
- soma de dois números negativos;
- soma com um dos elementos sendo zero;

Por exemplo, repare que os cenários que somam os valores 1+1 e 5+7, por exemplo, pertencem à mesma classe e portanto não precisam de testes para cada um destes cenários. Basta o desenvolvedor escolher um deles e escrever o teste.

Uma ótima maneira para garantir que não há testes repetidos é de novo apelar para um bom nome de teste. Por exemplo, se o nome do teste refere-se diretamente aos valores concretos do cenário, como *deveSomarUmMaisUm()*, onde o cenário é claramente a soma de (1 + 1), isso sugere ao desenvolvedor que crie um novo teste para a mesma classe de equivalência (a existência do método de teste *deveSomarDoisMaisDois()* não "fica estranho").

Ao contrário, caso o desenvolvedor opte por dar nomes que representam a classe de equivalência como, por exemplo, *deveSomarDoisNumerosPositivos()*, dificilmente o desenvolvedor criará o teste *deveSomarDoisNumerosPositivos_2()*.

```
public class CalculadoraTest {
  @Test
  public void deveSomarDoisNumerosPositivos() {
    assertEquals(4, new Calculadora().soma(2,2));
  }

  @Test
  public void deveSomarPositivoComNegativo() {
    assertEquals(4, new Calculadora().soma(6,-2));
  }

  @Test
  public void deveSomarNegativoComPositivo() {
    assertEquals(-4, new Calculadora().soma(-6,2));
  }

  @Test
  public void deveSomarDoisNumerosNegativos() {
    assertEquals(-4, new Calculadora().soma(-2,-2));
  }

  @Test
  public void deveSomarComZero() {
    assertEquals(4, new Calculadora().soma(0,4));
    assertEquals(4, new Calculadora().soma(4,0));
  }
}
```

Obviamente, encontrar todas as classes de equivalência não é um trabalho fácil, e por isso existe a gigante área de testes de software. Mas fato é que não é repetindo teste que o desenvolvedor garante a qualidade do software produzido.

6.5 Escrevendo boas asserções

Boas asserções garantirão que seus testes realmente falharão na hora certa. Mas, escrevê-los com qualidade envolve uma série de pequenos detalhes e o programador deve estar atento para eles.

Lembre-se que um bom teste é aquele que falhará quando preciso. E mais, ele mostrará o erro da forma mais clara possível. Um primeiro detalhe simples, mas importante, é a utilização do método *assertEquals()*, ou qualquer outro da família. Eles usualmente recebem 2 parâmetros. Como ele comparará se os dois objetos são iguais, então aparentemente a ordem em que os passamos não faz diferença.

```
assertEquals(a,b);
assertEquals(b,a);
```

No entanto, a ordem é importante para que o JUnit apresente a mensagem de erro corretamente. No momento em que um teste falha, o JUnit mostra uma mensagem parecida com essa: *java.lang.AssertionError: expected:<1> but was:<0>*, ou seja, esperava *X*, mas o valor calculado foi *Y*. Imagine se a mensagem acima viesse ao contrário: *java.lang.AssertionError: expected:<0> but was:<1>*. Isso só dificultaria a vida do programador na hora de entender a causa do problema.

Para que ele exiba a mensagem de erro correta, o desenvolvedor deve passar os parâmetros na ordem: o primeiro parâmetro é o valor esperado, o segundo parâmetro é o valor calculado. Geralmente o primeiro valor é "fixo" ou vem de uma variável declarada dentro do próprio método do teste; já o segundo, vem geralmente de um objeto que foi retornado pelo método sob teste.

Ao verificar que o objeto retornado pelo método de teste está correto, o desenvolvedor pode fazer uso de mais de um assert no mesmo método de teste. Em muitos casos, isso é obrigatório, aliás: Se o método sob teste retornou um novo objeto, é obrigação do teste garantir que todo seu conteúdo está correto.

Para isso, o desenvolvedor pode fazer uso de diversos asserts no mesmo objeto, um para cada atributo modificado, ou mesmo escrever apenas um, mas verificando o objeto inteiro através do método *equals()*, que é invocado pelo JUnit. O desenvolvedor deve escolher a alternativa que mais lhe agrada. Particularmente, prefiro a segunda opção.

```
// alternativa 1
Item item = itemQueVeioDoMetodoSobTeste();
assertEquals("Geladeira", item.getDescricao());
assertEquals(900.0, item.getValorUnitario());
assertEquals(1, item.getQuantidade());
assertEquals(900.0, item.getValorTotal());

// alternativa 2
Item item = itemQueVeioDoMetodoSobTeste();
Item itemEsperado = new Item("Geladeira", 1, 900.0);
assertEquals(itemEsperado, item);
```

Alguns desenvolvedores também gostam de deixar claro as pré-condições do cenário, e o fazem por meio de asserts. Por exemplo, se fossemos testar o método `adiciona()` do carrinho de compras, garantiríamos que após a invocação do comportamento, o item foi guardado na lista.

Para isso, verificaríamos o tamanho da lista, que seria 1, por exemplo. Mas, para que isso seja verdade, precisamos garantir que o carrinho esteja vazio nesse momento. Para garantir isso, uma asserção pode ser escrita antes do comportamento ser invocado.

```
@Test
public void deveAdicionarItens() {
    // garante que o carrinho está vazio
    assertEquals(0, carrinho.getItens().size());

    Item item = new Item("Geladeira", 1, 900.0);
    carrinho.adiciona(item);

    assertEquals(1, carrinho.getItens().size());
    assertEquals(item, carrinho.getItens().get(0));
}
```

Essa estratégia deve ser usada com parcimônia. O teste que contém asserts como esses pode ser mais difícil de ser lido.

Com exceção da necessidade de se escrever mais de 1 assert para garantir o conteúdo de objetos retornados pelo método sob teste, desenvolvedores devem evitar ao máximo fazer asserções em mais atributos do que deveria.

Fuja ao máximo de testes que validam mais de um comportamento. Testes devem ser curtos e testar apenas uma única responsabilidade da classe. Por exemplo, nos

testes da classe *CarrinhoDeCompras*, o teste que valida o método *maiorValor()* não deve validar o comportamento do método *adiciona()* e vice-versa.

Testes com duas responsabilidades tendem a ser mais complexos (afinal, precisam de mais linhas para montar um cenário que atenda aos dois comportamentos) e dão menos *feedback* para o desenvolvedor; se o teste falhar, qual das duas responsabilidades não está funcionando de acordo? O desenvolvedor precisará ler o código para entender. Além disso, o nome do método de teste pode ficar confuso, já que o desenvolvedor explicará o comportamento de duas funcionalidades em uma única frase.

6.6 Testando listas

Métodos que retornam listas também merecem atenção especial. Garantir só a quantidade de elementos da lista não é suficiente. Isso não garantirá que os objetos lá dentro estão como esperado. Portanto, o teste deve também validar o conteúdo de cada um dos objetos pertencentes à lista.

```
List<Item> lista = metodoQueDevolve2Itens();

assertEquals(2, lista.size());

assertEquals(100.0, lista.get(0).getValorUnitario());
assertEquals(200.0, lista.get(1).getValorUnitario());

// asserts nos outros atributos, como quantidade,
// etc, nos objetos dessa lista
```

Repare que às vezes seu método pode devolver uma lista com uma quantidade grande de elementos, tornando difícil a escrita dos asserts. Tente ao máximo montar cenários que facilitem seu teste, ou encontre um subconjunto de elementos nessa lista que garantam que o resto do conjunto estará correto e teste apenas esses elementos.

Mais de um assert no teste. O que isso significa?

Em meu doutorado, pesquiso muito sobre a relação do código de testes e a qualidade do código de produção. Em um de meus estudos, ainda em andamento, tenho verificado a relação entre a quantidade de asserts em um teste e qualidade do código que está sendo testado.

Apesar de um pouco imaturo, o estudo mostrou que, quando um método faz asserts em mais de uma instância de objetos diferentes, o código de produção que está sendo testado tende a ser pior (em termos de complexidade ciclomática e quantidade de linhas de código) do que métodos cujos testes fazem asserts em apenas uma única instância de objeto.

Portanto, essa pode ser mais um *feedback* que o teste lhe dá: se você está fazendo asserts em objetos diferentes no mesmo teste, isso pode ser um indício de código problemático.

6.7 Separando as Classes de Teste

Uma pergunta importante também é onde guardar as classes de teste. A sugestão principal é sempre separar suas classes de produção das classes de teste. No Eclipse, você pode criar "Source Folders" diferentes, e assim realmente isolar uns dos outros. Isso facilitará a vida do desenvolvedor na hora de gerar o pacote de produção, que não deve incluir os testes.

Uma outra questão importante é em qual pacote colocar as classes de teste. Aqui existem duas linhas principais de raciocínio.

Muitos desenvolvedores afirmam que as classes de teste devem ficar em pacotes idênticos às classes de produção. Dessa forma, fica fácil de encontrar onde está a classe de teste de determinada classe de produção.

Mas, muitos outros desenvolvedores afirmam que elas devem, na verdade, ficar em pacotes diferentes. Assim, o programador só conseguirá invocar os métodos públicos daquela classe, e não ficará tentado em testar métodos protegidos.

Me agrada a mistura de ambas. Eu mantenho meus códigos de teste sempre nos mesmos pacotes das classes de produção, e me forço a nunca testar nada que não seja público. As razões disso serão discutidas mais a frente.

Novamente reitero o ponto de que, independente da alternativa escolhida, se sua equipe está satisfeita com a decisão, mantenha.

6.8 Conclusão

Cuidar dos testes é primordial. Testes de baixa qualidade em algum momento atrapalharão a equipe de desenvolvimento. Por esse motivo, todas as boas práticas de código que desenvolvedores já usam em seus códigos de produção devem ser aplicadas ao código de teste. Test data builders, métodos de inicialização, reúso de código para evitar repetição, boas asserções, e etc, garantirão facilidade de leitura e evolução desses códigos.

Além disso, conforme discutido no capítulo anterior, buscaremos nos testes pequenas dicas sobre a qualidade do código de produção. Isso será difícil se o código do teste estiver bagunçado. Ou seja, se o desenvolvedor tenta manter o código de testes limpo, mas mesmo assim não consegue, há uma grande chance do código de produção ter problemas.

Capítulo 7

TDD e a Coesão

Classes que fazem muita coisa são difíceis de serem mantidas. Ao contrário, classes com poucas responsabilidades são mais simples e mais fáceis de serem evoluídas.

Uma classe coesa é justamente aquela que possui apenas uma única responsabilidade. Em sistemas orientados a objetos, a ideia é sempre buscar por classes coesas. Neste capítulo, discutiremos como TDD nos ajuda a encontrar classes com problemas de coesão e, a partir dessa informação, como refatorar o código.

7.1 Novamente o Problema do Cálculo de Salário

Relembrando o problema, é necessário calcular o salário dos funcionários da empresa a partir do seu cargo. Para isso, é necessário seguir as regras abaixo:

- Desenvolvedores possuem 20% de desconto caso seu salário seja maior do que R$ 3000,0. Caso contrário, o desconto é de 10%.
- DBAs e testadores possuem desconto de 25% se seus salários forem maiores do que R$ 2500,0. 15%, em caso contrário.

Já temos alguns testes escritos. Por exemplo, o teste abaixo garante que desenvolvedores com salário inferior a R$3000,00 recebam apenas 10% de desconto:

```
@Test
public void
deveCalcularSalarioParaDesenvolvedoresComSalarioAbaixoDoLimite() {

    CalculadoraDeSalario calculadora = new CalculadoraDeSalario();
    Funcionario desenvolvedor = new Funcionario
        ("Mauricio", 1500.0, Cargo.DESENVOLVEDOR);

    double salario = calculadora.calculaSalario(desenvolvedor);

    assertEquals(1500.0 * 0.9, salario, 0.00001);
}
```

A implementação atual, que resolvia o problema da regra de negócios para os desenvolvedores, parava aqui:

```
public double calculaSalario(Funcionario funcionario) {
    if(funcionario.getSalario() > 3000) {
        return funcionario.getSalario() * 0.8;
    }
    return funcionario.getSalario() * 0.9;
}
```

O problema começa a aparecer quando testamos um cargo diferente de desenvolvedor. Por exemplo, os testes abaixo validam o comportamento da classe para DBAs:

```
@Test
public void deveCalcularSalarioParaDBAsComSalarioAbaixoDoLimite() {

    CalculadoraDeSalario calculadora = new CalculadoraDeSalario();
    Funcionario dba = new Funcionario
        ("Mauricio", 1500.0, Cargo.DBA);

    double salario = calculadora.calculaSalario(dba);

    assertEquals(1500.0 * 0.85, salario, 0.00001);
}
```

```
@Test
public void deveCalcularSalarioParaDBAsComSalarioAcimaDoLimite() {

    CalculadoraDeSalario calculadora = new CalculadoraDeSalario();
    Funcionario dba = new Funcionario
        ("Mauricio", 4500.0, Cargo.DBA);

    double salario = calculadora.calculaSalario(dba);

    assertEquals(4500.0 * 0.75, salario, 0.00001);
}
```

Para fazê-los passar, é necessário algum condicional no código de produção para diferenciar o cálculo de desenvolvedores e DBAs, afinal as regras são diferentes. A implementação abaixo, por exemplo, resolve o problema e faz todos os testes passarem:

```
public double calculaSalario(Funcionario funcionario) {
    if(funcionario.getCargo().equals(Cargo.DESENVOLVEDOR)) {
        if(funcionario.getSalario() > 3000) {
            return funcionario.getSalario() * 0.8;
        }
        return funcionario.getSalario() * 0.9;
    }
    else if(funcionario.getCargo().equals(Cargo.DBA)) {
        if(funcionario.getSalario() < 2500) {
            return funcionario.getSalario() * 0.85;
        }
        return funcionario.getSalario() * 0.75;
    }

    throw new RuntimeException("Funcionario invalido");
}
```

A implementação dos testes para o cargo de testador é bem similar. Basta garantir o comportamento do sistema para funcionários que recebem acima e abaixo do limite. O código de produção também terá uma solução parecida. Basta adicionar um novo "if" para o cargo específico e a implementação será finalizada.

A implementação final seria algo do tipo:

```
public double calculaSalario(Funcionario funcionario) {
    if(funcionario.getCargo().equals(Cargo.DESENVOLVEDOR)) {
```

```
        if(funcionario.getSalario() > 3000) {
            return funcionario.getSalario() * 0.8;
        }
        return funcionario.getSalario() * 0.9;
    }
    else if(funcionario.getCargo().equals(Cargo.DBA) ||
    funcionario.getCargo().equals(Cargo.TESTADOR)) {
        if(funcionario.getSalario() < 2500) {
            return funcionario.getSalario() * 0.85;
        }
        return funcionario.getSalario() * 0.75;
    }

    throw new RuntimeException("Funcionario invalido");
}
```

Como o código está um pouco extenso, é possível refatorá-lo, extraindo algumas dessas linhas para métodos privados. Por exemplo:

```
public class CalculadoraDeSalario {

    public double calculaSalario(Funcionario funcionario) {
        if(funcionario.getCargo().equals(Cargo.DESENVOLVEDOR)) {
            return dezOuVintePorCentoDeDesconto(funcionario);
        }
        else if(funcionario.getCargo().equals(Cargo.DBA) ||
        funcionario.getCargo().equals(Cargo.TESTADOR)) {
            return quinzeOuVinteCincoPorCentoDeDesconto(funcionario);
        }

        throw new RuntimeException("Funcionario invalido");
    }

    private double
    quinzeOuVinteCincoPorCentoDeDesconto(Funcionario funcionario) {
        if(funcionario.getSalario() < 2500) {
            return funcionario.getSalario() * 0.85;
        }
        return funcionario.getSalario() * 0.75;
    }

    private double dezOuVintePorCentoDeDesconto(Funcionario funcionario){
```

```
        if(funcionario.getSalario() > 3000) {
            return funcionario.getSalario() * 0.8;
        }
        return funcionario.getSalario() * 0.9;
    }
}
```

Excelente. É hora de discutir melhor sobre nosso código de testes e código de produção.

Escrever mais de um teste de uma só vez?

No código acima, mostrei diretamente dois métodos de teste de uma só vez. A pergunta é: você deve fazer isso? Afinal, isso não é TDD.

Nesse momento, minha sugestão é para que você escreva teste a teste, veja cada um falhar, e faça cada um passar na hora certa. Apenas por questões de didática e facilidade de leitura, colei ambos de uma só vez.

Mais para frente, discutiremos sobre quando ou não usar TDD.

7.2 Ouvindo o *feedback* dos testes

Todos os testes passam e a implementação resolve o problema atual. Mas será que o código está simples e fácil de ser mantido? O código de produção nesse momento possui alguns problemas graves em termos de evolução.

O primeiro deles é a complexidade. Veja que com apenas 2 cargos implementados, escrevemos 2 ifs, com outros ifs dentro deles. Quanto maior a quantidade de condicionais, mais complexo o código fica. Essa complexidade, aliás, tem um nome: complexidade ciclomática [32]. Complexidade ciclomática, de maneira simplificada, é o número de diferentes caminhos que seu método pode executar. Por exemplo, quando adicionamos um if no código, automaticamente geramos dois diferentes caminhos: um caminho onde a condição do if é verdadeiro, e outro caminho onde a condição é falsa. Quanto maior a combinação de ifs, fors, e coisas do gênero, maior será esse número. Portanto, do ponto de vista de implementação, esse código é bastante complexo.

Agora, do ponto de vista de design, o código atual apresenta um problema ainda pior: sempre que criarmos um novo cargo no sistema (e isso é razoavelmente simples, basta adicionar um novo item no enum), é necessário fazer essa alteração tam-

bém na calculadora de salário. Em um sistema real, essas "dependências implícitas" são geralmente uma das causa de sistemas apresentarem constantes problemas, pois o desenvolvedor nunca sabe em quais classes ele precisa mexer para propagar uma nova regra de negócio. Imagine que a cada novo cargo criado, o desenvolvedor precisasse atualizar outras 20 classes? Em algum momento ele esqueceria, afinal nada "força" ele a fazer essa alteração. Códigos frágeis como esse são comuns em implementações geralmente procedurais.

Mas o teste, de certa, já estava avisando sobre esses problemas. Veja a bateria atual de testes:

```
deveCalcularSalarioParaDesenvolvedoresComSalarioAbaixoDoLimite()
deveCalcularSalarioParaDesenvolvedoresComSalarioAcimaDoLimite()
deveCalcularSalarioParaDBAsComSalarioAbaixoDoLimite()
deveCalcularSalarioParaDBAsComSalarioAcimaDoLimite()
deveCalcularSalarioParaTestadoresComSalarioAbaixoDoLimite()
deveCalcularSalarioParaTestadoresComSalarioAcimaDoLimite()
```

Repare que essa bateria de testes tende a ser infinita. Quanto mais cargos aparecerem, mais testes serão criados nessa classe. Se a classe de testes possui muitos testes, isso quer dizer que a classe de produção possui muitas responsabilidades. E, em sistemas orientados a objetos, sabemos que classes devem ser coesas, conter apenas uma única responsabilidade. Essa classe tende a ter milhares de responsabilidades. Quando uma classe de testes tende a crescer indefinidamente, isso pode ser um sinal de má coesão na classe de produção.

Além disso, repare no nome dos testes: deve calcular salário **para desenvolvedores** com salario abaixo do limite. Repare que esse "para desenvolvedores" indica que o comportamento da classe muda de acordo com uma característica do objeto que o método recebe como parâmetro (no caso, o cargo do funcionário). Isso nos leva a crer que todo novo cargo precisará de um novo teste **para o cargo novo**. Qualquer variação dessa no nome do teste, ("para X", "se X", "como X", etc) pode indicar um problema na abstração dessa classe. Em um bom sistema orientado a objetos, os comportamentos evoluem naturalmente, geralmente por meio da criação de novas classes, e não pela alteração das classes já existentes.

Ou seja, de bater o olho no código de testes, encontramos duas maneiras baratas de encontrar problemas na classe de produção: classes de teste que não param de crescer e nomes de testes que evidenciam a falta de uma abstração melhor para o problema. O trabalho do desenvolvedor nesse momento é entender o porquê isso

está acontecendo, e eventualmente refatorar a classe de produção para resolver esse problema.

> **Complexidade ciclomática e quantidade de testes**
>
> Já que a complexidade ciclomática nos diz a quantidade de diferentes caminhos que um método tem, e já que sabemos que devemos testar todos os diferentes comportamentos do nosso método, é possível inferir que a complexidade ciclomática tem uma relação direta com a quantidade de testes de um método.
>
> Ou seja, quanto maior a complexidade ciclomática, maior a quantidade de testes necessários para garantir seu comportamento. Portanto, um código com alta complexidade ciclomática, além de ser confuso, ainda exige um alto esforço para ser testado.

7.3 Testes em métodos privados?

Veja agora o método *quinzeOuVinteCincoPorCentoDeDesconto()*. Ele é responsável por calcular quinze ou vinte por cento de desconto de acordo com o salário do funcionário. Um método não complicado, mas que possui sim alguma regra de negócio envolvida. É necessário testá-lo. A pergunta é como fazer para testá-lo, afinal ele é um método privado, o que o impossibilita de ser invocado diretamente por um teste.

Uma discussão grande em torno da comunidade de software é justamente sobre a necessidade de se testar métodos privados [6]. Alguns desenvolvedores acreditam que métodos como esse devem ser testados, e optam por fazer uso de frameworks que possibilitam a escrita de testes para métodos privados (através de uso de *reflection*, o framework consegue invocar esse método). Outros acreditam que você não deve testá-lo diretamente, mas sim indiretamente, através de um método público que faz uso. No caso, igual feito nos testes acima (o método *calculaSalario()* invoca o método *quinzeOuVinteCincoPorCentoDeDesconto()*).

Novamente a discussão sobre o *feedback* que o teste dá ao desenvolvedor. **Se o desenvolvedor sente a necessidade de testar um método privado de maneira isolada, direta, ou seja, sem passar por um método público que faz uso dele, muito provavelmente é porque esse método privado faz muita coisa**.

Métodos privados são geralmente subdivisões de um método público maior. Optamos por eles para facilitar a leitura desse método público. Mas muitas vezes cria-

mos métodos privados que trabalham demais e possuem uma responsabilidade tão bem definida que poderia constituir uma nova classe.

Ao perceber isso, o desenvolvedor deve se perguntar se o método está realmente no lugar certo. Talvez movê-lo para alguma outra classe, ou até mesmo criar uma nova justamente para acomodar esse comportamento faça sentido.

Portanto, evite testar métodos privados. Leve isso como um *feedback* sobre a qualidade da sua classe. Extraia esse comportamento para uma nova classe ou mova-o para uma classe já existente. Transforme-o em um método público que faça sentido e aí teste-o decentemente.

7.4 Resolvendo o Problema da Calculadora de Salário

Sempre que houver uma separação dos comportamentos em várias pequenas classes, com o objetivo de torná-las mais coesas, é necessário uni-las novamente para se obter o comportamento maior, esperado.

Para fazer isso de maneira elegante, é necessário algum conhecimento em orientação a objetos. Muitos padrões de projeto [19], por exemplo, tem como objetivo unir, de maneira clara, classes que precisam trabalhar juntas.

Para esse novo problema, por exemplo, podemos começar por resolver o problema dos métodos privados. Vamos extraí-los para classes específicas, cada uma responsável por uma regra. Como elas são semelhantes, ambas classes implementarão a mesma interface *RegraDeCalculo*. Essa implementação se parece com o padrão de projeto Strategy:

```
public interface RegraDeCalculo {
    double calcula(Funcionario f);
}

public class DezOuVintePorCento implements RegraDeCalculo {

    public double calcula(Funcionario funcionario) {
        if(funcionario.getSalario() > 3000) {
            return funcionario.getSalario() * 0.8;
        }
        return funcionario.getSalario() * 0.9;
    }
```

```
}

public class QuinzeOuVinteCincoPorCento implements RegraDeCalculo {

    public double calcula(Funcionario funcionario) {
        if(funcionario.getSalario() < 2500) {
            return funcionario.getSalario() * 0.85;
        }
        return funcionario.getSalario() * 0.75;
    }

}
```

Repare que cada regra de cálculo agora está em sua classe específica, bem definida. Testar essa classe agora é fácil. Teremos 2 ou 3 testes e pronto. Diferentemente da antiga classe `CalculadoraDeSalario`, a bateria de ambas as regras de cálculo não tendem a crescer infinitamente. Isso nos mostra que essas classes são razoavelmente coesas.

Agora, para forçar o desenvolvedor a sempre definir uma regra de cálculo para todo e qualquer novo cargo, podemos forçá-lo a decidir isso na própria enumeração de Cargo. Enumerações em Java são quase como classes. Elas podem ter construtores e possuir métodos. Faremos então todo cargo conter uma regra de cálculo:

```
public enum Cargo {
    DESENVOLVEDOR(new DezOuVintePorCento()),
    DBA(new QuinzeOuVinteCincoPorCento()),
    TESTADOR(new QuinzeOuVinteCincoPorCento());

    private final RegraDeCalculo regra;

    Cargo(RegraDeCalculo regra) {
        this.regra = regra;
    }

    public RegraDeCalculo getRegra() {
        return regra;
    }
}
```

Por fim, com todas as responsabilidades bem definidas, a classe `CalculadoraDeSalario` agora ficará bem mais simples. Veja:

```
public class CalculadoraDeSalario {
    public double calculaSalario(Funcionario funcionario) {
        return funcionario.getCargo().getRegra().calcula(funcionario);
    }
}
```

Ela somente repassa a chamada para a regra de cálculo. Nosso código agora está muito mais orientado a objetos. Se um novo cargo aparecer, precisamos apenas adicionar no enum. Se uma nova regra de cálculo aparecer, basta criarmos uma nova classe que implementa a interface certa. Todas as classes são pequenas e fáceis de serem testadas. Nosso sistema evolui facilmente e todo código escrito é simples.

> **Preciso testar a classe CalculadoraDeSalario?**
>
> A calculadora de salário agora contém apenas uma linha, que repassa para a regra fazer o cálculo.
>
> Talvez essa classe não precise nem existir mais no sistema. Mas, caso exista, talvez não seja necessário testá-la. O teste só garantiria a delegação para a regra específica. Se um dia a calculadora ficar mais complexa, aí sim testes farão mais sentido para ela.

Ferreira fala

A refatoração do código de exemplo da calculadora de salário demonstrou muito bem como a atenção à coesão nos leva a um design melhor das nossas classes. Será que podemos ir além? Outro mau cheiro no código é a repetição, no caso, da lógica de decisão a respeito do desconto a ser aplicado. Observamos que os métodos `calcula()` das duas implementações de regra de cálculo são muito similares. Podemos abstrair essa lógica fazendo da `RegraDeCalculo` uma classe abstrata:

```
public abstract class RegraDeCalculo {

  public double calcula(Funcionario funcionario) {
    if (funcionario.getSalario() > limite()) {
      return funcionario.getSalario() * porcentagemAcimaDoLimite();
    }
    return funcionario.getSalario() * porcentagemBase();
  }
```

```
  protected abstract int limite();

  protected abstract double porcentagemAcimaDoLimite();

  protected abstract double porcentagemBase();

}
```

E modificando as implementações para definirem concretamente os métodos que estabelecem os parâmetros específicos. Por exemplo, a classe DezOuVintePorCento ficaria:

```
public class DezOuVintePorCento extends RegraDeCalculo {
  @Override
  protected double porcentagemBase() {
    return 0.9;
  }

  @Override
  protected double porcentagemAcimaDoLimite() {
    return 0.8;
  }

  @Override
  protected int limite() {
    return 3000;
  }

}
```

Assim eliminaríamos a duplicação; esse tipo de estrutura é chamada de **Template Method**. Mas será mesmo que o design ficou melhor? Não existe uma resposta fácil a essa pergunta, é um *trade-off* entre flexibilidade e abstração, mas na minha opinião eu diria que não, que nesse caso o nível de abstração obtido não vale o custo.

7.5 O que olhar no teste em relação a coesão?

Como visto acima, os testes podem nos avisar sobre problemas de coesão em nossas classes. Lembre-se que classes não coesas fazem muita coisa; testar "muita coisa" não é fácil. Escrever testes deve ser uma tarefa fácil.

Quando um único método necessita de diversos testes para garantir seu comportamento, o método em questão provavelmente é complexo e/ou possui diversas responsabilidades. Códigos assim possuem geralmente diversos caminhos diferentes e tendem a alterar muitos atributos internos do objeto, obrigando o desenvolvedor a criar muitos testes, caso queira ter uma alta cobertura de testes. A esse padrão, dei o nome de **Muitos Testes Para Um Método**.

Também pode ser entendido quando o desenvolvedor escreve muitos testes para a classe como um todo. Classes que expõem muitos métodos para o mundo de fora também tendem a possuir muitas responsabilidades. Chamo esse padrão de **Muitos Testes Para Uma Classe**.

Outro problema de coesão pode ser encontrado quando o programador sente a necessidade de escrever cenários de teste muito grandes para uma única classe ou método. É possível inferir que essa necessidade surge em códigos que lidam com muitos objetos e fazem muita coisa. Nomeei esse padrão de **Cenário Muito Grande**.

A vontade de testar um método privado também pode ser considerado um indício de problemas de coesão. Métodos privados geralmente servem para transformar o método público em algo mais fácil de ler. Ao desejar testá-lo de maneira isolada, o programador pode ter encontrado um método que possua uma responsabilidade suficiente para ser alocada em uma outra classe. A esse padrão, chamo de **Testes em Método Que Não É Público**.

Lembre-se que esses padrões não dão qualquer certeza sobre o problema. Eles são apenas indícios. O programador, ao encontrar um deles, deve visitar o código de produção e realmente comprovar se existe um problema de coesão. De maneira mais geral, lembre-se que qualquer dificuldade na escrita de um cenário ou o trabalho excessivo para se criar cenários para testar uma única classe ou método pode indicar problemas de coesão.

Classes devem ser simples. E classes simples são testadas de forma simples. A relação entre código de produção e código de teste é realmente forte. A busca pela simplicidade no teste nos leva ao encontro de um código de produção mais simples. E assim que deve ser.

7.6 Conclusão

Os testes podem nos dar dicas boas e baratas sobre o nível de coesão das nossas classes. Obviamente, você desenvolvedor deve estar atento a isso e melhorar o código de acordo.

Mas lembre-se que essas dicas são apenas heurísticas para problemas de coesão. Não é possível garantir que toda a vez que você se deparar com uma classe com muitos testes, ela apresenta problemas de coesão. Como tudo em engenharia de software, tudo depende de um contexto. Ou seja, mesmo com esses pequenos padrões, não conseguimos tirar o lado criativo e racional do desenvolvedor. Você deve avaliar caso a caso.

Capítulo 8

TDD e o Acoplamento

No capítulo anterior, discutimos a relação entre a prática de TDD e classes coesas. Agora é necessário olhar o outro lado da balança: o acoplamento. Dizemos que uma classe está acoplada a outra quando existe alguma relação de dependência entre elas. Por exemplo, em nosso código anterior, a classe `Funcionario` é acoplada com a enumeração `Cargo`. Ou sejam ela depende da enumeração. Isso quer dizer que mudanças na enumeração podem impactar de forma negativa a classe.

Classes altamente coesas e pouco acopladas são difíceis de serem projetadas. Neste capítulo, discutiremos como TDD ajuda o desenvolvedor a encontrar problemas de acoplamento no seu projeto de classes.

8.1 O Problema da Nota Fiscal

O processo de geração de nota fiscal é geralmente complicado. Envolve uma série de cálculos de acordo com as diversas características do produto vendido. Mas, mais complicado ainda pode ser o fluxo de negócio após a geração da nota: enviá-la para

um sistema maior, como um SAP, enviar um e-mail para o cliente com a nota fiscal gerada, persistir as informações na base de dados, e assim por diante.

De maneira simplificada, uma possível representação da Nota Fiscal e Pedido pode ser semelhante a abaixo:

```
public class Pedido {

    private String cliente;
    private double valorTotal;
    private int quantidadeItens;

    public Pedido(String cliente, double valorTotal,
                  int quantidadeItens) {
        this.cliente = cliente;
        this.valorTotal = valorTotal;
        this.quantidadeItens = quantidadeItens;
    }

    public String getCliente() {
        return cliente;
    }

    public double getValorTotal() {
        return valorTotal;
    }

    public int getQuantidadeItens() {
        return quantidadeItens;
    }
}

public class NotaFiscal {

    private String cliente;
    private double valor;
    private Calendar data;

    public NotaFiscal(String cliente, double valor, Calendar data) {
        this.cliente = cliente;
        this.valor = valor;
        this.data = data;
```

```
    }

    public String getCliente() {
        return cliente;
    }

    public double getValor() {
        return valor;
    }

    public Calendar getData() {
        return data;
    }

}
```

Imagine que hoje esse processo consiste em gerar a nota, persistir no banco de dados e enviá-la por e-mail. No capítulo passado, discutimos sobre coesão e classes com responsabilidades específicas. Para o problema proposto, precisamos de 3 diferentes classes: uma classe responsável por enviar o e-mail, outra responsável por persistir as informações na base de dados, e por fim, uma classe responsável pelo cálculo do valor da nota fiscal.

Para facilitar o entendimento do exemplo, a implementação das classes que se comunicam com o banco de dados e com o SAP serão simplificadas. Veja o código abaixo:

```
public class SAP {
    public void envia(NotaFiscal nf) {
        // envia NF para o SAP
    }
}

public class NFDao {
    public void persiste(NotaFiscal nf) {
        // persiste NF
    }
}
```

Agora é possível iniciar a escrita da classe `GeradorDeNotaFiscal`, responsável por calcular o valor final da nota e, em seguida, disparar a sequência do processo de negócio.

Imagine que a regra para cálculo da nota fiscal seja simples: o valor da nota deve ser o valor do produto subtraído de 6%. Ou seja, 94% do valor total. Esse é um problema parecido com os anteriores. Começando pelo teste:

```
@Test
public void deveGerarNFComValorDeImpostoDescontado() {
    GeradorDeNotaFiscal gerador = new GeradorDeNotaFiscal();
    Pedido pedido = new Pedido("Mauricio", 1000, 1);

    NotaFiscal nf = gerador.gera(pedido);

    assertEquals(1000 * 0.94, nf.getValor(), 0.0001);
}
```

Fazer o teste passar é trivial. Basta instanciarmos uma nota fiscal com 6% do valor a menos do valor do pedido. Além disso, a data da nota fiscal será a data de hoje:

```
public class GeradorDeNotaFiscal {
    public NotaFiscal gera(Pedido pedido) {
        return new NotaFiscal(
                pedido.getCliente(),
                pedido.getValorTotal() * 0.94,
                Calendar.getInstance());
    }
}
```

O teste passa. O próximo passo é persistir essa nota fiscal. A classe `NFDao` já existe. Basta fazermos uso dela. Dessa vez, sem escrever o teste antes, vamos ver como ficaria a implementação final:

```
public class GeradorDeNotaFiscal {
    public NotaFiscal gera(Pedido pedido) {
        NotaFiscal nf = NotaFiscal(
                pedido.getCliente(),
                pedido.getValorTotal() * 0.94,
                Calendar.getInstance());

        new NFDao().persiste(nf);

        return nf;
```

```
    }
}
```

A grande pergunta agora é: como testar esse comportamento? Acessar o banco de dados e garantir que o dado foi persistido? Não parece uma boa ideia.

8.2 Mock Objects

A ideia de um teste de unidade é realmente testar a classe de maneira isolada, sem qualquer interferência das classes que a rodeiam. A vantagem desse isolamento é conseguir um maior *feedback* do teste em relação a classe sob teste; em um teste onde as classes estão integradas, se o teste falha, qual classe gerou o problema?

Além disso, testar classes integradas pode ser difícil para o desenvolvedor. Em nosso exemplo, como garantir que o elemento foi realmente persistido? Seria necessário fazer uma consulta posterior ao banco de dados, garantir que a linha está lá, limpar a tabela para que na próxima vez que o teste for executado, os dados já existentes na tabela não atrapalhem o teste, e assim por diante.

É muito trabalho. E na verdade, razoavelmente desnecessário. Persistir o dado no banco de dados é tarefa da classe `NFDao`. A tarefa da classe `GeradorDeNotaFiscal` é somente invocar o DAO. Não há porquê os testes da `GeradorDeNotaFiscalTest` garantirem que o dado foi persistido com sucesso; isso seria tarefa da classe `NFDaoTest`.

Nesse caso, uma alternativa seria simular o comportamento do %NFDao `no momento do teste do gerador de nota fiscal. Ou seja, queremos criar um clone de` NFDaoao banco de dados. Classes que simulam o comportamento de outras são chamadas de **mock objects**, ou objetos dublês.

Um framework de mock facilita a vida do desenvolvedor que deseja criar classes falsas para seus testes. Um mock pode ser bem inteligente. É possível ensinar o mock a reagir da maneira que queremos quando um método for invocado, descobrir a quantidade de vezes que o método foi invocado pela classe de produção, e assim por diante.

No caso de nosso exemplo, precisamos garantir que o método `dao.persiste()` foi invocado pela classe `GeradorDeNotaFiscal`. Nosso teste deve então criar o mock e garantir que o método esperado foi invocado.

Aqui utilizaremos o framework conhecido como Mockito [28]. Com ele, criar

mocks e validar o esperado é fácil. Sua API é bem simples e fluente. Veja o teste abaixo, onde criamos o mock e depois validamos que o método foi invocado:

```
@Test
public void devePersistirNFGerada() {
    // criando o mock
    NFDao dao = Mockito.mock(NFDao.class);

    GeradorDeNotaFiscal gerador = new GeradorDeNotaFiscal();
    Pedido pedido = new Pedido("Mauricio", 1000, 1);

    NotaFiscal nf = gerador.gera(pedido);

    // verificando que o método foi invocado
    Mockito.verify(dao).persiste(nf);
}
```

O problema é que se rodarmos o teste, ele falha. Isso acontece porque, apesar de termos criado o mock, o `GeradorDeNotaFiscal` não faz uso dele. Repare que na implementação, ele instancia um `NFDao`. É necessário que o gerador seja mais esperto: ele deve usar o mock no momento do teste, e a classe `NFDao` quando o código estiver em produção.

Uma solução para isso é receber a classe pelo construtor. Dessa forma, é possível passar o mock ou a classe concreta. E em seguida, fazer uso do DAO recebido:

```
public class GeradorDeNotaFiscal {
    private NFDao dao;

    public GeradorDeNotaFiscal(NFDao dao) {
        this.dao = dao;
    }

    public NotaFiscal gera(Pedido pedido) {
        NotaFiscal nf = new NotaFiscal(
                pedido.getCliente(),
                pedido.getValorTotal() * 0.94,
                Calendar.getInstance());

        dao.persiste(nf);

        return nf;
```

```
    }

}
```

Agora, basta passar o mock para a classe sob teste e o comportamento do DAO será simulado:

```
@Test
public void devePersistirNFGerada() {
    NFDao dao = Mockito.mock(NFDao.class);

    GeradorDeNotaFiscal gerador = new GeradorDeNotaFiscal(dao);
    Pedido pedido = new Pedido("Mauricio", 1000, 1);

    NotaFiscal nf = gerador.gera(pedido);

    Mockito.verify(dao).persiste(nf);
}
```

Podemos fazer a mesma coisa com os próximos passos da geração da nota fiscal, como enviar para o SAP, enviar por e-mail, e etc. Basta passar mais uma dependência pelo construtor, criar outro mock e garantir que o método foi enviado. Com o SAP, por exemplo:

```
@Test
public void deveEnviarNFGeradaParaSAP() {
    NFDao dao = Mockito.mock(NFDao.class);
    SAP sap = Mockito.mock(SAP.class);

    GeradorDeNotaFiscal gerador = new GeradorDeNotaFiscal(dao, sap);
    Pedido pedido = new Pedido("Mauricio", 1000, 1);

    NotaFiscal nf = gerador.gera(pedido);

    Mockito.verify(sap).envia(nf);
}

public class GeradorDeNotaFiscal {
    private NFDao dao;
    private final SAP sap;
```

```
    public GeradorDeNotaFiscal(NFDao dao, SAP sap) {
        this.dao = dao;
        this.sap = sap;
    }

    public NotaFiscal gera(Pedido pedido) {
        NotaFiscal nf = new NotaFiscal(
                pedido.getCliente(),
                pedido.getValorTotal() * 0.94,
                Calendar.getInstance());

        dao.persiste(nf);
        sap.envia(nf);

        return nf;
    }

}
```

TDD no estilo londrino

Muitas pessoas conhecem a prática de TDD e mockar as dependências como "TDD ao estilo londrino". Muitas das discussões importantes na área de mock objects surgiram por lá. Famosos autores como Steve Freeman e Nat Pryce (ambos britânicos) são fãs dessa abordagem.

Independente do nome, me agrada muito a utilização de mock objects durante a escrita dos testes. Quando estou criando uma classe, mocks me ajudam a pensar somente no que ela vai fazer e como ela vai interagir com as outras classes do sistema. Nesse momento, me preocupo muito pouco com as outras classes e foco somente do que espero da classe atual. Sem a utilização de mocks, isso seria um pouco mais trabalhoso.

8.3 Dependências explícitas

Repare que por uma necessidade do teste, optamos por receber as dependências da classe pelo construtor. Isso é, na verdade, uma boa prática quando se pensa em orientação a objetos.

Em primeiro lugar o desenvolvedor deixa as dependências da classe explícitas.

Basta olhar seu construtor, e ver quais classes são necessárias para que ela faça seu trabalho por completo. Em segundo lugar, ao receber a dependência pelo construtor, a classe facilita sua extensão. Por meio de polimorfismo, o desenvolvedor pode, a qualquer momento, optar por passar alguma classe filho da classe que é recebida (ou, no caso de uma interface, outra classe a implementa), mudando/evoluindo seu comportamento.

Novamente retomo a frase de que um código fácil de ser testado possui características interessantes do ponto de vista de design. Explicitar as dependências é uma necessidade quando se pensar em testes de unidade, afinal essa é a única maneira de se passar um mock para a classe.

8.4 Ouvindo o *feedback* dos testes

A classe `GeradorDeNotaFiscal` tem um problema grave de evolução. Imagine se após a geração da nota houvesse mais 10 atividades para serem executadas, similares a persistir no banco de dados e enviar para o SAP. A classe passaria a receber uma dezena de dependências no construtor, se tornando altamente acoplada. Classes como essas são chamadas de **God Classes**, pois elas geralmente contém pouca regra de negócio, e apenas coordenam o processo de várias classes juntas.

Repare que, na prática, essa a balança tradicional em sistemas orientados a objetos. No capítulo anterior, tínhamos uma única classe, pouco acoplada, mas que continha todas as regras de negócio dentro dela, tornando-o difícil de ser lida e testada. Por fim, optamos por separar os comportamentos em pequenas classes. Já neste capítulo, os comportamentos já estão separados, já que a classe `NFDao` contém a lógica de acesso a dados, a classe `SAP` contém a comunicação com o SAP, e assim por diante. Mas é necessário juntar essas pequenas classes para realizar o comportamento total esperado e, ao fazermos, caímos no problema do acoplamento.

Os testes escritos podem nos dar dicas importantes sobre problemas de acoplamento. Um primeiro *feedback* importante é justamente a quantidade de mock objects em um teste. Imagine se nossa classe realmente fizesse uso de outras 10 classes; o teste teria 10 mock objects. Apesar da não necessidade nesse cenário específico, em muitos testes existe a necessidade de ensinar o mock object a reagir de acordo com o esperado. Logo, o teste gastaria muitas linhas simplesmente para montar todos os objetos dublês necessários para o teste.

Além disso, o "mau uso" de mock objects pode ser indício de problemas com a abstração e o excesso de acoplamento. Veja o teste

deveEnviarNFGeradaParaSAP, por exemplo. Repare que para esse teste, a interação com o dublê do NFDao pouco importa. Ou seja, a classe contém uma dependência que interessa apenas a um subconjunto dos testes, mas não para todos eles. Por que isso aconteceria? Provavelmente existe uma outra maneira de desenhar isso de forma a melhorar o gerenciamento de dependências dessa classe e diminuir seu acoplamento.

8.5 Classes estáveis

Uma das conhecidas vantagens do TDD é que ele "diminui o acoplamento", ou "ajuda a diminuir o acoplamento". A grande pergunta é como ele faz. Suponha o código abaixo, que lê um XML e escreve o conteúdo por uma porta serial:

```
public class Copiadora {
  public void copiar() {
    LeitorDeXML leitor = new LeitorDeXML();
    EscritorPelaSerial escritor = new EscritorPelaSerial();
    while (leitor.temCaracteres()) {
      escritor.escreve(leitor.leCaracteres());
    }
  }
}
```

Veja que essa classe é fortemente acoplada com duas classes: LeitorDeXML e EscritorPelaSerial. Se quantificarmos isso, podemos dizer que o acoplamento dessa classe é igual a 2 (já que ela está acoplada a 2 classes).

Ao praticar TDD para a criação da classe Copiadora, o desenvolvedor provavelmente focará primeiramente na classe Copiadora e no que ela deve fazer, esquecendo os detalhes da implementação do leitor e do escritor. Para conseguir esquecer desses detalhes, o desenvolvedor provavelmente criaria interfaces que representariam as ações esperadas de leitura e escrita:

```
@Test
public void deveLerEEnviarOConteudoLido() {
    Escritor e = Mockito.mock(Escritor.class);
    Leitor l = Mockito.mock(Leitor.class);

    Mockito.when(l.temCaracteres()).thenReturn(true, false);
    Mockito.when(l.leCaracteres()).thenReturn("mauricio");
```

```
    Copiadora c = new Copiadora(l, e);

    c.copiar();

    Mockito.verify(e).escreve("mauricio");
}

public interface Leitor {
  boolean temCaracteres();
  String leCaracteres();
}

public interface Escritor {
  void escreve(String conteudo);
}
```

Repare que o teste cria dois mocks, escritor e leitor. Em seguida, define o comportamento esperado pelo dublê do leitor: o método `temCaracteres()` deve devolver "verdadeiro" e "falso", nessa ordem, e `leCaracteres()` devolvendo o texto "mauricio". A seguir, cria a `Copiadora`, passando os dublês, invoca o comportamento sob teste, e por fim verifica que o escritor recebeu a instrução para escrever a palavra esperada.

Repare que para esse teste pouco importa como funcionará a classe que lê e a classe que escreve. Ao praticar TDD, isso é comum. Nos preocupamos menos com as classes que a classe atual interagirá, mas sim apenas com a interface que elas devem prover. Não há forma melhor de representar esses contratos do que utilizando interfaces.

Mas qual a diferença entre ambos os códigos do ponto de vista do acoplamento? Afinal, nas duas implementações, a classe `Copiadora` continua dependendo de 2 outras classes (ou interfaces).

A diferença é em relação a *estabilidade* da segunda classe [24]. Repare que a segunda implementação depende apenas de interfaces e não de classes concretas. O real problema do alto acoplamento é que as dependências de uma classe podem sofrer mudanças, propagando-as para a classe principal. Logo, quanto maior a dependência, mais instável é a classe, ou seja, maior a chance dela sofrer uma mudança.

Interfaces, por sua vez, tendem a ser classes que mudam muito pouco por várias razões. Primeiro porque elas não contém detalhes de implementação, atributos ou qualquer outra coisa que tenda a mudar. Segundo porque geralmente interfaces contém diversas implementações embaixo delas e o desenvolvedor tende a não alterá-la, pois sabe que se o fizer, precisará alterar em todas as implementações. Logo, se interfaces não mudam (ou mudam muito pouco), acoplar-se com elas pode não ser tão problemático.

Veja no exemplo acima. As classes `LeitorDeXML` e `EscritorPelaSerial` provavelmente dependem de outras classes (a leitora de XML deve fazer uso de alguma biblioteca para tal, bem como a que escreve pela serial), possuem atributos e muito código para fazer seu serviço. A chance delas mudarem é alta, e portanto tendem a ser instáveis. Ao contrário, as interfaces `Leitor` e `Escritor` não dependem de nada, não possuem código e classes as implementam. Logo, são estáveis, o que significa que depender delas não é uma ideia tão má.

Podemos dizer então que uma boa dependência é uma dependência com um módulo (ou classe) estável, e uma má dependência é uma dependência com um módulo (ou classe) instável. Portanto, se houver a necessidade de se acoplar com alguma outra classe, que seja com uma classe razoavelmente estável.

TDD faz com que o programador acople suas classes com módulos geralmente mais estáveis! TDD força o desenvolvedor a pensar apenas no que você espera das outras classes, sem pensar ainda em uma implementação concreta. Esses comportamentos esperados acabam geralmente se transformando em interfaces, que frequentemente se tornam estáveis.

Import estático do Mockito

Nos testes, estamos escrevendo linhas como `Mockito.mock()` ou `Mockito.verify()`. Na prática, os desenvolvedores geralmente optam por fazer o import estático da classe `Mockito` e, dessa forma, fazer uso dos métodos `mock()`, `verify()` e etc, diretamente no código do teste, simplesmente para facilitar a leitura.

8.6 Resolvendo o Problema da Nota Fiscal

Para gerenciar melhor a dependência da classe `GeradorDeNotaFiscal`, é necessário fazer com que ela dependa de módulos estáveis; uma interface é a melhor candidata. Repare que toda ação executada após a geração da nota, independente de qual for, precisa receber apenas a nota fiscal gerada para fazer seu trabalho. Logo, é possível criar uma interface para representar todas elas e fazer com que `SAP`, `NFDao` ou qualquer outra ação a ser executada após a geração da nota, implemente-a:

```
public interface AcaoAposGerarNota {
    void executa(NotaFiscal nf);
}
```

Veja que `AcaoAposGerarNota` tende a ser estável: é uma interface e possui algumas classes concretas que a implementa.

O próximo passo agora é fazer com que o gerador de nota fiscal dependa de uma lista dessas ações, e não mais de classes concretas. Uma sugestão é receber essa lista pelo construtor, já deixando a dependência explícita:

```
public class GeradorDeNotaFiscal {

    private final List<AcaoAposGerarNota> acoes;

    public GeradorDeNotaFiscal(List<AcaoAposGerarNota> acoes) {
        this.acoes = acoes;
    }

    // ...
}
```

Basta agora fazer com que o gerador, após criar a nota, invoque todas as ações que estão na lista:

```
public NotaFiscal gera(Pedido pedido) {
    NotaFiscal nf = new NotaFiscal(
            pedido.getCliente(),
            pedido.getValorTotal() * 0.94,
            Calendar.getInstance());

    for(AcaoAposGerarNota acao : acoes) {
        acao.executa(nf);
    }

    return nf;
}
```

Repare agora que a classe `GeradorDeNotaFiscal` não está mais fortemente acoplada a uma ação concreta, mas sim a uma lista qualquer de ações. A chance dela sofrer uma alteração propagada por uma dependência é bem menor. Além do mais, a evolução dessa classe passa a ser natural: basta passar mais itens na lista de ações, que ela as executará. Não há mais a necessidade de alterar a classe para adicionar um novo comportamento. Em orientação a objetos, a ideia de estendermos o comportamento de uma classe sem alterar seu código é conhecido como **Princípio do Aberto-Fechado**, ou OCP (Open-Closed Principle).

Para testarmos, precisamos apenas garantir que as ações da lista, quaisquer que sejam, serão executadas pelo gerador:

```
@Test
public void deveInvocarAcoesPosteriores() {
    AcaoAposGerarNota acao1 = Mockito.mock(AcaoAposGerarNota.class);
    AcaoAposGerarNota acao2 = Mockito.mock(AcaoAposGerarNota.class);

    List<AcaoAposGerarNota> acoes = Arrays.asList(acao1, acao2);

    GeradorDeNotaFiscal gerador = new GeradorDeNotaFiscal(acoes);
    Pedido pedido = new Pedido("Mauricio", 1000, 1);

    NotaFiscal nf = gerador.gera(pedido);

    Mockito.verify(acao1).executa(nf);
    Mockito.verify(acao2).executa(nf);
}
```

Toda essa refatoração, no fim, foi disparada por um *feedback* do nosso teste: o mau uso ou o uso excessivo de mock objects. O teste avisando o problema de design aliado ao conhecimento de orientação a objetos, proporcionou uma melhoria significativa na classe.

8.7 Testando métodos estáticos

Outro tipo de acoplamento, difícil de ser testado, é o acoplamento com métodos estáticos. Acabamos de ver que para simular o comportamento das classes, é necessário fazer uso de objetos dublês e, de alguma forma, fazer com que a classe receba e utilize esses dublês. No caso de métodos estáticos, não há como recebê-los pelo construtor.

Em nosso exemplo, imagine que a data da nota fiscal nunca possa ser no fim de semana. Se a nota for gerada no sistema, o sistema deve empurrar sua data para segunda-feira. Em Java, para pegarmos a data atual do sistema, fazemos uso do método estático `Calendar.getInstance()`. Como simular seu comportamento e escrever um teste para a geração de uma nota fiscal em um sábado?

Nesses casos, a sugestão é sempre criar uma abstração para facilitar o teste. Ou seja, ao invés de fazer uso direto do método estático, criar uma classe/interface, responsável por devolver a hora atual, e que seja possível de ser mockada. Podemos, por exemplo, criar a interface `Relogio` e a implementação concreta `RelogioDoSistema`:

```
public interface Relogio {
    Calendar hoje();
}

public class RelogioDoSistema implements Relogio {
    public Calendar hoje() {
        return Calendar.getInstance();
    }
}
```

O gerador, por sua vez, passa a fazer uso de um relógio para pegar a data atual:

```
public class GeradorDeNotaFiscal {

    private final List<AcaoAposGerarNota> acoes;
    private final Relogio relogio;

    public GeradorDeNotaFiscal(List<AcaoAposGerarNota> acoes,
        Relogio relogio) {
        this.acoes = acoes;
        this.relogio = relogio;
    }
```

```
    // construtor sem Relogio para não
    // quebrar o resto do sistema
    public GeradorDeNotaFiscal(List<AcaoAposGerarNota> acoes) {
        this(acoes, new RelogioDoSistema());
    }

    public NotaFiscal gera(Pedido pedido) {
        NotaFiscal nf = new NotaFiscal(
                pedido.getCliente(),
                pedido.getValorTotal() * 0.94,
                relogio.hoje());

        for(AcaoAposGerarNota acao : acoes) {
            acao.executa(nf);
        }

        return nf;
    }
}
```

Recebendo esse `Relogio` pelo construtor, o teste é natural. Basta montar o cenário esperado, e verificar a saída do método de acordo com o esperado.

Métodos estáticos são problemáticos. Além de dificultarem a evolução da classe, afinal o desenvolvedor não consegue fazer uso de polimorfismo e fazer uso de uma implementação diferente desse método, eles ainda dificultam o teste.

Evite ao máximo criar métodos estáticos. Só o faça quando o método for realmente simples. Tente sempre optar por interfaces e implementações concretas, fáceis de serem dubladas. Ao lidar com APIs de terceiros, crie abstrações em cima delas única e exclusivamente para facilitar o teste.

Lembre-se: **você pode tomar decisões de design pensando exclusivamente na testabilidade**. Seu código, acima de tudo, deve ser confiável, e para isso deve ser testado.

8.8 TDD e a constante criação de interfaces

Imagine que o problema do gerador de notas fiscais torne-se um pouco mais complicado. Para calcular o valor do imposto, devemos olhar o valor do pedido. Essa tabela contém uma série de faixas de valor, cada uma com uma porcentagem associada. Por exemplo, valores entre zero e R$999 possuem 2% de imposto, valores entre

R$1000,00 e R$2999,00 possuem 6% de imposto, e assim por diante. Imagine que essa tabela é razoavelmente grande e está armazenada em um arquivo XML. O gerenciador, com essa porcentagem em mãos, deve apenas multiplicar pelo valor total do pedido.

Para o gerenciador de nota fiscal, pouco importa como a tabela faz seu trabalho, mas sim o valor que ela retorna. A experiência nos diz que a lógica responsável por essa tabela deve estar em uma classe separada, e o gerenciador deve fazer uso dela.

Podemos deixar isso explícito no teste, passando uma dependência `Tabela` (que ainda nem existe) para o construtor do gerenciador. Além disso, podemos exigir que essa tabela tenha sido consultado pelo gerenciador. O Mockito nos ajudará nisso:

```
@Test
public void deveConsultarATabelaParaCalcularValor() {

    // mockando uma tabela, que ainda nem existe
    Tabela tabela = Mockito.mock(Tabela.class);

    // definindo o futuro comportamento "paraValor",
    // que deve retornar 0.2 caso o valor seja 1000.0
    Mockito.when(tabela.paraValor(1000.0)).thenReturn(0.2);

    List<AcaoAposGerarNota> nenhumaAcao = Collections.emptyList();
    GeradorDeNotaFiscal gerador =
        new GeradorDeNotaFiscal(nenhumaAcao, tabela);
    Pedido pedido = new Pedido("Mauricio", 1000, 1);

    NotaFiscal nf = gerador.gera(pedido);

    // garantindo que a tabela foi consultada
    Mockito.verify(tabela).paraValor(1000.0);
    assertEquals(1000 * 0.2, nf.getValor(), 0.00001);

}
```

Nesse momento o teste não compila, pois não existe a classe `Tabela`. Podemos nesse momento optar por começar a trabalhar nessa classe e deixar o gerenciador de lado. Mas se fizermos isso, perderemos o foco. Nesse momento, não importa como a tabela fará o trabalho dela; mas sabemos exatamente o que esperar dela: uma porcentagem de acordo com um valor.

Podemos então definir uma interface para representar esse comportamento que será futuramente implementado por alguma classe:

```
public interface Tabela {
    double paraValor(double valor);
}
```

Com essa interface bem definida, podemos fazer o teste passar, recebendo a tabela no construtor e fazendo uso dela no método `gera()`:

```
public GeradorDeNotaFiscal(List<AcaoAposGerarNota> acoes,
    Relogio relogio, Tabela tabela) {
        this.acoes = acoes;
        this.relogio = relogio;
        this.tabela = tabela;
}

public NotaFiscal gera(Pedido pedido) {
    NotaFiscal nf = new NotaFiscal(
            pedido.getCliente(),
            pedido.getValorTotal() *
                            tabela.paraValor(pedido.getValorTotal()),
            relogio.hoje());

    for(AcaoAposGerarNota acao : acoes) {
        acao.executa(nf);
    }

    return nf;
}
```

Com essa interface agora, podemos criar mais testes que fazem uso da tabela, sem na verdade nos preocuparmos com a implementação, pois já conhecemos o contrato dela. Esse tipo de procedimento é muito comum durante a prática de TDD. Ao criar classes, percebemos que é necessário dividir o comportamento em classes diferentes. Nesse momento, é comum criarmos uma interface para representar o comportamento esperado e continuar a criação e testes daquela classe sem nos preocuparmos em como cada dependência fará seu trabalho.

Essa nova maneira de programar ajuda inclusive a criar contratos mais simples de serem implementados, afinal as interfaces conterão apenas os comportamentos

simples, diretos e necessários para que a classe principal faça seu trabalho. Em nosso exemplo, após acabar a implementação do gerador, o próximo passo agora seria criar a classe `TabelaDoGoverno`, por exemplo, que implementa a interface `Tabela` e descobre a porcentagem de acordo com a faixa de valores. Talvez essa classe precise fazer uso de alguma outra classe. Não há problema: criaríamos novamente uma outra interface para representar o comportamento esperado e continuaríamos a trabalhar na tabela.

Portanto, ao praticar TDD, o programador foca apenas no que ele precisa das outras classes. Isso faz com ele crie contratos estáveis e simples de serem implementados. Veja novamente como o teste e o foco nele fez com que pensássemos no projeto de classes.

Estilo londrino na prática

É assim que funciona o estilo londrino de TDD. As interfaces vão emergindo a medida que são necessárias. Os testes acabam por deixar explícito como funciona a interação entre essas várias classes.

Nesse momento, o uso de mock objects torna-se fundamental para facilitar o teste.

8.9 O que olhar no teste em relação ao acoplamento?

O uso abusivo de objetos dublês para testar uma única classe indica que a classe sob teste possui problemas de acoplamento. É possível deduzir que uma classe que faz uso de muitos objetos dublês depende de muitas classes, e portanto, tende a ser uma classe instável. A esse padrão, dei o nome de **Objetos Dublê em Excesso**.

A criação de objetos dublês que não são utilizados em alguns métodos de testes é outro *feedback* importante. Isso geralmente acontece quando a classe é altamente acoplada, e o resultado da ação de uma dependência não interfere na outra. Quando isso acontece, o programador acaba por escrever conjuntos de testes, sendo que alguns deles lidam com um subconjunto dos objetos dublês, enquanto outros testes lidam com o outro subconjunto de objetos dublês. Isso indica um alto acoplamento da classe, que precisa ser refatorada. A esse padrão dei o nome de **Objetos Dublê Não Utilizados**.

Quando o desenvolvedor começa o teste e percebe que a interface pública da classe não está amigável, pode indicar que abstração corrente não é clara o suficiente e poderia ser melhorada. A esse padrão, chamei de **Interface Não Amigável**.

A falta de abstração geralmente também faz com que uma simples mudança precise ser feita em diferentes pontos do código. Quando uma mudança acontece e o programador é obrigado a fazer a mesma alteração em diferentes testes, isso indica a falta de uma abstração correta para evitar a repetição desnecessária de código. A esse padrão dei o nome de **Mesma Alteração Em Diferentes Testes**. Analogamente, o programador pode perceber a mesma coisa quando ele começa a criar testes repetidos para entidades diferentes. Chamei esse padrão de **Testes Repetidos Para Entidades Diferentes**.

8.10 Conclusão

Neste capítulo, vimos padrões de *feedback* que os testes nos dão em relação ao acoplamento da classe. A grande maioria deles está totalmente relacionado ao mau uso de mock objects.

Discutimos também que para um melhor gerenciamento de dependências, classes devem sempre tentar depender de módulos estáveis, diminuindo assim as chances da propagação de mudanças para a classe principal. Isso é alcançado através do bom uso de orientação a objetos e interfaces.

A balança acoplamento e coesão é difícil de ser lidada. Classes coesas são fáceis de serem lidas e testadas, mas para que o sistema se comporte como o esperado, é necessário juntar todas essas pequenas classes. Nesse momento é onde deixamos o lado do acoplamento se perder. Infelizmente é impossível criar um sistema cujas classes são todas altamente coesas a pouco acopladas. Mas espero que ao final desses dois capítulos, eu tenha mostrado como atacar ambos os problemas e criar classes que balanceiam entre esses dois pontos-chave da orientação a objetos.

Capítulo 9

TDD e o Encapsulamento

Já discutimos sobre coesão e acoplamento nos capítulos anteriores, e vimos como os testes podem nos dar informações valiosas sobre esses dois pontos. Um próximo ponto importante em sistemas orientados a objetos é o **encapsulamento**.

Encapsular é esconder os detalhes de **como** a classe realiza sua tarefa; as outras classes devem conhecer apenas **o quê** ela faz. Ao não encapsular corretamente regras de negócios em classes específicas, desenvolvedores acabam por criar código repetido, e espalhar as regras de negócio em diferentes partes do sistema.

Neste capítulo, discutiremos como os testes podem nos ajudar a encontrar regras de negócio que não estão bem encapsuladas.

9.1 O Problema do Processador de Boleto

É comum que uma fatura possa ser paga por meio de diferentes boletos. Para resolver esse problema, um processador de boletos passa por todos os boletos pagos para uma fatura e simplesmente faz o vínculo de ambos.

Antes de começarmos, suponha a existência das classes `Fatura`, `Boleto` e `Pagamento`. Uma `Fatura` contém uma lista de `Pagamento`, que por sua vez, armazena um valor e uma forma da pagamento (boleto, cartão de crédito, etc). Um `Boleto` contém apenas o valor pago do boleto.

De antemão, já é possível imaginar alguns cenários para esse processador:

- Usuário pagou com apenas um boleto.
- Usuário utilizou mais de um boleto para pagar.

Começando pelo primeiro cenário, mais simples, temos o teste abaixo, que cria um único boleto, invoca o processador, e ao final garante que o pagamento foi criado na fatura:

```
class ProcessadorDeBoletosTest {
    @Test
    public void deveProcessarPagamentoViaBoletoUnico() {
        ProcessadorDeBoletos processador = new ProcessadorDeBoletos();

        Fatura fatura = new Fatura("Cliente", 150.0);
        Boleto b1 = new Boleto(150.0);
        List<Boleto> boletos = Arrays.asList(b1);

        processador.processa(boletos, fatura);

        assertEquals(1, fatura.getPagamentos().size());
        assertEquals(150.0,
                fatura.getPagamentos().get(0).getValor(), 0.00001);
    }
}
```

A implementação para fazer o teste passar é simples:

```
public class ProcessadorDeBoletos {

    public void processa(List<Boleto> boletos, Fatura fatura) {
        Boleto boleto = boletos.get(0);
        Pagamento pagamento = new Pagamento(boleto.getValor(),
            MeioDePagamento.BOLETO);
        fatura.getPagamentos().add(pagamento);
    }
}
```

Agora, o próximo cenário é garantir que o processador de boletos consegue processar um usuário que fez o pagamento por mais de um boleto:

```
@Test
public void deveProcessarPagamentoViaMuitosBoletos() {
    ProcessadorDeBoletos processador = new ProcessadorDeBoletos();

    Fatura fatura = new Fatura("Cliente", 300.0);
    Boleto b1 = new Boleto(100.0);
    Boleto b2 = new Boleto(200.0);
    List<Boleto> boletos = Arrays.asList(b1, b2);

    processador.processa(boletos, fatura);

    assertEquals(2, fatura.getPagamentos().size());
    assertEquals(100.0,
            fatura.getPagamentos().get(0).getValor(), 0.00001);
    assertEquals(200.0,
            fatura.getPagamentos().get(1).getValor(), 0.00001);
}
```

Para fazer o teste passar, basta navegar pela lista de boletos e criar um pagamento para cada um deles:

```
public class ProcessadorDeBoletos {

    public void processa(List<Boleto> boletos, Fatura fatura) {
        for(Boleto boleto : boletos) {
            Pagamento pagamento = new Pagamento(boleto.getValor(),
                MeioDePagamento.BOLETO);
            fatura.getPagamentos().add(pagamento);
        }
    }
}
```

Com o processador de boletos já funcionando, o próximo passo agora é marcar a fatura como paga, caso o valor dos boletos pagos seja igual ou superior ao valor da fatura. Ou seja, o sistema deve se comportar da seguinte maneira para os cenários abaixo:

- Se o usuário pagar um único boleto com valor inferior ao da fatura, ela não deve ser marcada como paga.

- Se o usuário pagar um único boleto com valor superior ao da fatura, ela deve ser marcada como paga.
- Se o usuário pagar um único boleto com valor igual ao da fatura, ela deve ser marcada como paga.
- Se o usuário pagar vários boletos e a soma deles for inferior ao da fatura, ela não deve ser marcada como paga.
- Se o usuário pagar vários boletos e a soma deles for superior ao da fatura, ela deve ser marcada como paga.
- Se o usuário pagar vários boletos e a soma deles for igual ao da fatura, ela deve ser marcada como paga.

Apesar da quantidade de cenários ser grande, a implementação não é tão complicada assim. Precisamos guardar a soma de todos os boletos pagos e, ao final, verificar se ela é maior ou igual ao valor da fatura. Em caso positivo, basta marcar a fatura como paga.

Novamente começando pelo cenário mais simples:

```
@Test
public void deveMarcarFaturaComoPagaCasoBoletoUnicoPagueTudo() {
    ProcessadorDeBoletos processador = new ProcessadorDeBoletos();

    Fatura fatura = new Fatura("Cliente", 150.0);
    Boleto b1 = new Boleto(150.0);
    List<Boleto> boletos = Arrays.asList(b1);

    processador.processa(boletos, fatura);

    assertTrue(fatura.isPago());
}
```

Fazendo o teste passar da maneira que discutimos anteriormente:

```
public void processa(List<Boleto> boletos, Fatura fatura) {
    double valorTotal = 0;

    for(Boleto boleto : boletos) {
        Pagamento pagamento = new Pagamento(boleto.getValor(),
```

```
            MeioDePagamento.BOLETO);
        fatura.getPagamentos().add(pagamento);

        valorTotal += boleto.getValor();
    }

    if(valorTotal >= fatura.getValor()) {
        fatura.setPago(true);
    }
}
```

Com todos os testes verdes, é hora de discutirmos sobre a implementação feita até então.

Devo escrever os outros testes?

Mesmo que a implementação pareça já resolver todos os cenários propostos, é importante que esses testes sejam automatizados.

Lembre-se que amanhã a implementação poderá mudar. Como garantir que a nova implementação funcionará para todos os casos que a implementação atual funciona? Todo código que você acabou de escrever parece simples e funcional, mas não se esqueça que ele ficará lá para sempre e será mantido por muitas outras pessoas.

Portanto, não se deixe enganar. Escreva os testes e garanta que nenhuma futura evolução quebrará o que já funciona hoje.

9.2 Ouvindo o *feedback* dos testes

Apesar da classe acima atender a regra de negócio, o código produzido não é dos melhores. O que aconteceria com o sistema caso precisássemos criar agora um processador de cartão de crédito? Seria necessário repetir a mesma lógica de marcar uma fatura como paga lá. Ou seja, a cada nova forma de pagamento, trechos de código seriam replicados entre classes. E não há necessidade para se discutir os problemas de código repetido.

O princípio ferido aqui é justamente o encapsulamento. As classes devem ser responsáveis por manter o seu próprio estado; elas é quem devem conhecer as suas próprias regras de negócio. Quando a regra não é seguida, pedaços da regra de ne-

gócio são espalhadas pelo código. É exatamente isso o que aconteceu no exemplo: a regra de marcar uma fatura como paga está fora da classe `Fatura`.

O teste, por sua vez, nos dá dicas sobre esse problema. Veja por exemplo qualquer um dos testes escritos na classe `ProcessadorDeBoletoTest`. Todos eles fazem asserções na classe `Fatura`. A pergunta é: por que os testes de uma classe fazem asserções somente em outras classes?

Responder essa pergunta pode ser difícil. Essa valiosa dica pode estar nos avisando sobre possíveis problemas de encapsulamento na classe que recebe a asserção. Neste caso, a classe `Fatura` não está encapsulando bem as suas regras de negócio.

9.3 Tell, Don't Ask e Lei de Demeter

Classes como a `ProcessadorDeBoletos`, que "conhecem demais" sobre o comportamento de outras classes não são bem vistas em sistemas orientados a objetos. É comum ouvir o termo **intimidade inapropriada**, já que a classe conhece detalhes que não deveria de alguma outra classe.

Repare a implementação do método `processa()` em relação ao processo de marcar a fatura como paga. Veja que ele faz uma pergunta para a fatura (pergunta o valor dela), e depois, com a resposta em mãos, toma uma ação (marca ou não a fatura como paga). Em códigos orientados a objetos, geralmente dizemos que as classes não devem fazer perguntas e tomar decisões baseados nas respostas, mas sim **mandar** o objeto executar uma ação, e ele por conta própria tomar a decisão certa. Essa ideia é conhecida por **Tell, Don't Ask**.

Em nosso exemplo, a ideia seria mandar a classe fatura se marcar como paga, caso necessário, afinal ela deve ser a responsável por manter seu próprio estado. Essa ação poderia ser tomada, por exemplo, no momento em que adicionamos um novo pagamento.

Se a `Fatura` tivesse um método `adicionaPagamento(Pagamento p)`, que, além de adicionar o pagamento na fatura, ainda somasse os valores pagos e marcasse a fatura como paga se necessário, o problema estaria resolvido. Não teríamos mais o `for` do lado de fora da classe. Essa regra estaria dentro da classe, encapsulada.

Uma outra possível maneira de descobrir por possíveis problemas de encapsulamento é contando a quantidade de métodos invocados em uma só linha dentro de um único objeto. Por exemplo, imagine o seguinte código:

```
a.getB().getC().getD().fazAlgumaCoisa();
```

Para invocar o comportamento desejado, partimos de "a", pegamos "b", "c" e "d". Relembre agora a discussão sobre acoplamento. Quando uma classe depende da outra, mudanças em uma classe podem se propagar para a classe principal. Perceba agora que a classe que contém acima não está acoplada somente a classe do tipo do atributo "a", mas indiretamente também aos tipos de "b", "c" e "d". Qualquer mudança na estrutura deles pode impactar na classe principal.

Isso pode ser considerado também um problema de encapsulamento. A classe do tipo "a" está expondo demais sua estrutura interna. O "mundo de fora" sabe que ela lá dentro tem "b", que por sua vez, tem "c", "d", e assim por diante. Se o comportamento `fazAlgumaCoisa()` deve realmente ser invocado, pode-se fazer algo como um método dentro de "a" que esconde o processo de invocá-lo:

```
a.fazAlgumaCoisa();

class A {
    public void fazAlgumaCoisa() {
        this.getB().getC().fazAlgumaCoisa();
    }
}
```

Diminuir a quantidade de iterações com os objetos, ou seja, navegar menos dentro deles, é o que conhecemos por **Lei de Demeter**. Ela nos diz justamente isso: tente nunca conversar com classes que estão dentro de classes; para isso, crie um método que esconda esse trabalho pra você. Dessa forma, você encapsula melhor o comportamento esperado e ainda reduz o acoplamento.

Devo seguir a Lei de Demeter à risca?

Como tudo em engenharia de software, não. A Lei de Demeter, apesar de muito interessante, às vezes pode mais atrapalhar do que ajudar.

Geralmente não ligo para linhas como `pessoa.getEndereco().getRua()`, pois estamos apenas pegando dados de um objeto. Não faz sentido criar um método `pessoa.getRua()`, ou `pessoa.getX()` para todo dado que representa uma pessoa. É simplesmente trabalhoso demais.

Essa é geralmente a minha regra: para exibição, aceito não seguir a regra. Mas para invocar um comportamento, geralmente penso duas vezes antes de não seguir.

9.4 Resolvendo o Problema do Processador de Boletos

Conforme já discutido acima, é necessário levar a regra de negócios da marcação da fatura como paga para dentro da própria classe `Fatura`. Seguindo a ideia do método `adicionaPagamento()`, faremos ele adicionar o pagamento e marcar a fatura como paga caso necessário:

```
public void adicionaPagamento(Pagamento pagamento) {
    this.pagamentos.add(pagamento);

    double valorTotal = 0;
    for(Pagamento p : pagamentos) {
        valorTotal += p.getValor();
    }

    if(valorTotal >= this.valor) {
        this.pago = true;
    }
}
```

O processador de boletos volta a ficar com o código enxuto:

```
public void processa(List<Boleto> boletos, Fatura fatura) {
    for(Boleto boleto : boletos) {
        Pagamento pagamento = new Pagamento(boleto.getValor(),
            MeioDePagamento.BOLETO);
        fatura.adicionaPagamento(pagamento);
    }
}
```

Veja que agora ele não conhece detalhes da classe `Fatura`. Isso é responsabilidade da própria fatura. Não há mais intimidade inapropriada. O desenvolvedor deve agora mover os testes que garantem a marcação de uma fatura como paga para dentro da classe `FaturaTest`, já que isso não é mais um comportamento do processador de boletos.

9.5 O que olhar no teste em relação ao encapsulamento?

Testes que lidam demais com outros objetos ao invés de lidar com o objeto sob teste podem estar avisando o desenvolvedor em relação a problemas de encapsulamento. A própria não utilização da Lei de Demeter, tanto nos testes quanto no código de produção, também pode avisar sobre os mesmos problemas.

Isso é comum em bateria de testes de classes anêmicas [5]. Um modelo anêmico é aquele onde classes contém apenas atributos ou apenas métodos. Classes que contém atributos apenas representam as entidades, enquanto outras classes, que contém apenas métodos, realizam ações sobre eles. Esse tipo de projeto (que faz com que o código se pareça mais com um código procedural do que com um código orientado a objetos) deve ser evitado ao máximo.

9.6 Conclusão

Neste capítulo, discutimos o ponto que faltava para encerrarmos a discussão sobre *feedback* dos testes em relação a pontos importantes em projetos orientados a objetos: coesão, acoplamento e encapsulamento.

Muito provavelmente você desenvolvedor encontrará outros padrões de *feedback* que o teste pode dar. Siga seu instinto, experiência e conhecimento. O importante é ter código de qualidade ao final.

Capítulo 10

Testes de Integração e TDD

Até o momento, escrevemos apenas testes de unidade. Será que faz sentido escrever testes ou até mesmo praticar TDD para outros níveis de teste?

Neste capítulo, discutiremos sobre o mau uso de mocks, testes para acesso a dados, testes de integração, e quando não fazê-los.

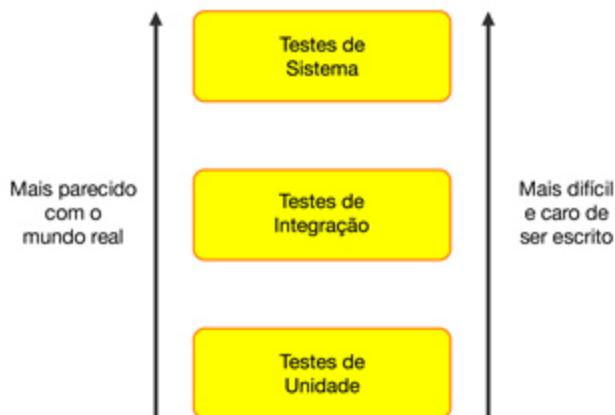
10.1 Testes de unidade, integração e sistema

Podemos escrever um teste de diferentes maneiras, de acordo com o que esperamos obter dele. Todos os testes que escrevemos até agora são conhecidos por **testes de unidade**.

Um teste de unidade é aquele que garante que uma classe funciona, de maneira isolada ao resto do sistema. Ou seja, testamos o comportamento dela sem se preocupar com o comportamento das outras classes. Uma vantagem dos testes de unidade é que eles são fáceis de serem escritos e rodam muito rápido. A desvantagem deles é que eles não simulam bem a aplicação no mundo real. No mundo real, temos as mais diversas classes trabalhando juntas para produzir o comportamento maior esperado.

Se quisermos um teste que se pareça com o mundo real, ou seja, que realmente teste a aplicação do ponto de vista do usuário, é necessário escrever o que chamamos de **teste de sistema**. Um teste de sistema é aquele que é idêntico ao executado pelo usuário da aplicação. Se sua aplicação é uma aplicação web, esse teste deve então subir o browser, clicar em links, submeter formulários, e etc. A vantagem desse tipo de teste é que ele consegue encontrar problemas que só ocorrem no mundo real, como problemas de integração entre a aplicação e banco de dados, e etc. O problema é que eles geralmente são mais difíceis de serem escritos e levam muito mais tempo para serem executados.

No entanto, muitas vezes queremos testar não só uma classe, mas também não o sistema todo; queremos testar a integração entre uma classe e um sistema externo. Por exemplo, classes DAO (responsáveis por fazer toda a comunicação com o banco de dados) devem ser testadas para garantir que as consultas SQL estão escritas corretamente, mas de maneira isolada as outras classes do sistema. Esse tipo de teste, que garante a integração entre 2 pontos da aplicação, é conhecido por **teste de integração**.

O desenvolvedor deve fazer uso dos diferentes níveis de teste para garantir qualidade do seu sistema. Mas deve sempre ter em mente que, quanto mais parecido com o mundo real, mais difícil e caro o teste será.

10.2 Quando não usar mocks?

O primeiro passo para que um teste deixe de ser exclusivamente de unidade e passe a ser de integração é não usar mocks e passar dependências concretas para a classe sob teste.

Muitos desenvolvedores, inclusive, defendem que um bom teste nunca faz uso de mocks. O argumento deles é de que mocks acabam por "esconder" possíveis problemas que só seriam pegos na integração. O argumento de sempre em relação a testes de unidade e testes de integração. O ponto não é descobrir se devemos ou não usar mocks, mas sim **quando ou não** usá-los.

Veja, por exemplo, um dos testes implementados para a calculadora de salário:

```
@Test
public void
    deveCalcularSalarioParaDesenvolvedoresComSalarioAbaixoDoLimite() {

    CalculadoraDeSalario calculadora = new CalculadoraDeSalario();
    Funcionario desenvolvedor = new
        Funcionario("Mauricio", 1500.0, Cargo.DESENVOLVEDOR);

    double salario = calculadora.calculaSalario(desenvolvedor);

    assertEquals(1500.0 * 0.9, salario, 0.00001);
}
```

Esse teste garante o comportamento da classe `CalculadoraDeSalario`. Idealmente, gostaríamos de testá-la independente do comportamento das outras classes. Mas veja que no teste, instanciamos um `Funcionario`. Usamos a classe concreta e não fizemos uso de mock. Por quê?

Geralmente classes que representam entidades, serviços, utilitários, ou qualquer outra coisa que encosta em infra estrutura, não são mockadas. Elas são classes Java pura e simples e mocká-las só irá dar mais trabalho ao desenvolvedor.

Ao tomar essa decisão, diminuímos a qualidade do retorno desse teste, afinal ele pode falhar não por culpa da calculadora de salário, mas sim por culpa do funcionário. Apesar de não ser o melhor dos mundos, é uma troca justa entre produtividade e *feedback*.

Opte por mockar classes que lidam com infra estrutura e que tornariam seu teste muito complicado. Por exemplo, relembre nosso gerador de nota fiscal. A nota era persistida em um banco de dados e depois enviada para o SAP. Preparar tanto o

banco quanto o SAP para receber o teste não é fácil. Portanto, simular a interação de ambas as classes é uma boa ideia.

Use mocks também quando sua classe lida com interfaces. Em nossa versão final do gerador de nota fiscal, criamos a interface `AcaoAposGerarNota`. Nesses casos, o mais simples talvez seja mockar a interface ao invés de criar uma implementação concreta "simples", somente para o teste.

Por outro lado, um ponto negativo do uso de mock objects é o alto acoplamento criado entre o código de teste e o código de produção. Já discutimos o conceito de encapsulamento no capítulo anterior: uma classe deve esconder a maneira na qual ela implementa determinada regra de negócio. Quando um teste faz uso de um mock, esse teste passa a ter uma "intimidade inapropriada" com a implementação; ele passa a saber quais métodos serão invocados, e como a classe deve reagir de acordo com o resultado.

Veja o código de produção e teste abaixo, responsável por calcular o imposto de um pedido, de acordo com uma tabela de preços:

```
@Test
public void deveCalcularImpostoParaPedidosSuperioresA2000Reais() {
    TabelaDePrecos tabela = mock(TabelaDePrecos.class);

    // ensinando o mock a devolver 1 caso o método
    // pegaParaValor seja invocado com o valor 2500.0
    when(tabela.pegaParaValor(2500.0)).thenReturn(0.1);

    Pedido pedido = new Pedido(2500.0);
    CalculadoraDeImposto calculadora = new CalculadoraDeImposto(tabela);

    double valor = calculadora.calculaImposto(pedido);

    assertEquals(2500 * 0.1, valor, 0.00001);
}

public double calculaImposto(Pedido p) {
    double taxa = tabela.pegaParaValor(p.getValor());
    return p.getValor() * taxa;
}
```

Veja que o teste `deveCalcularImpostoParaPedidosSuperioresA2000Reais` sabe exatamente qual método será invocado no código de produção. Isso quer dizer

que qualquer mudança na implementação do método pode fazer o teste quebrar. Ou seja, quanto maior o uso de mocks, mais delicado e frágil seu teste fica.

Muitos desenvolvedores, quando são apresentados a ideia de mock objects, passam a mockar todos os objetos nos seus testes. Muitos testes escritos com esse pensamento passam a ser inúteis, pois acabam "testando o mock". No capítulo seguinte, veremos um exemplo disso. Mas um primeiro ponto de alerta é: se sua bateria de testes só faz asserções em objetos dublês, talvez essa bateria não esteja lhe dando o *feedback* necessário.

Portanto, faça uso de mock objects quando utilizar a instância concreta da classe for complexo ou trabalhoso. Quando usar a classe concreta não for diminuir o *feedback* dos seus testes e nem dificultar a escrita dele, então use a classe concreta.

10.3 Testes em DAOs

Testar classes que fazem uso de DAOs é simples; basta fazermos uso de mocks. Mas o DAO é uma classe, e que também merece ser testada. Testar um DAO é fundamental. É muito comum errarmos em consultas SQL, esquecermos de algum campo no momento da inserção ou atualizar campos que não deveriam ser atualizados.

Mas como toda classe que lida com infra estrutura, testá-la é um pouco mais complicado. É razoavelmente óbvio que para se testar um DAO, é necessário comunicar-se de verdade com um banco de dados real. Nada de simulações ou mocks, afinal essa é a única maneira de garantir que a consulta foi escrita e executada com sucesso pelo banco de dados.

Imagine um DAO qualquer, por exemplo, o abaixo, que contém um método que salva o produto, outro que busca um produto pelo seu id, e outro que devolve somente os produtos ativos. Veja que esse DAO faz uso da Session do Hibernate (uma maneira elegante de acessar dados no mundo Java):

```
public class ProdutoDao {

    private final Session session;

    public ProdutoDao(Session session) {
        this.session = session;
    }

    public void adiciona(Produto produto) {
        session.save(produto);
```

```
    }

    public Produto porId(int id) {
        return (Produto) session.load(Produto.class, id);
    }

    public List<Produto> ativos() {
        return session
            .createQuery("from Produto p where p.ativo = true")
            .list();
    }
}
```

Precisamos testar cada um desses métodos. Vamos começar pelo método `adiciona()`. Como todo teste, ele conterá as mesmas 3 partes: cenário, ação e validação. O cenário e ação são similares aos dos nossos testes anteriores. Basta criarmos um produto e invocar o método `adiciona`:

```
@Test
public void deveAdicionarUmProduto() {
    ProdutoDao dao = new ProdutoDao(session);

    Produto produto = new Produto("Geladeira", 150.0);

    dao.adiciona(produto);

    // como validar?
}
```

O problema é justamente como validar que o dado foi inserido com sucesso. Precisamos garantir que o elemento foi salvo no banco de dados. A única maneira é justamente fazendo uma consulta de seleção no banco de dados e verificando o produto lá. Para isso, podemos usar o próprio método `porId()` do DAO, e verificar se o objeto salvo é igual ao produto criado:

```
@Test
public void deveAdicionarUmProduto() {
    ProdutoDao dao = new ProdutoDao(session);

    Produto produto = new Produto("Geladeira", 150.0);
```

```
    dao.adiciona(produto);

    // buscando no banco pelo id
    // para ver se está igual ao produto do cenário
    Produto salvo = dao.porId(produto.getId());
    assertEquals(salvo, produto);
}
```

Ainda temos um problema. Nosso `ProdutoDao` faz uso de uma `Session` do Hibernate. Todo DAO tem sua forma de conectar com o banco de dados. É necessário passar uma sessão concreta, que bata em um banco de dados de teste, para que o teste consiga executar.

Mockando a Connection/Session?

Já ouvi falar de desenvolvedores que testam DAOs mockando a Connection, Session (do Hibernate), ou qualquer que seja a sua forma de acesso ao banco de dados.

Minha pergunta é: em uma classe cuja única responsabilidade é interagir com o sistema externo, qual a vantagem de isolá-la desse sistema? Não faz sentido. Classes que lidam com infra estrutura devem ser testadas com a infra estrutura real que lidarão. É trabalhoso, mas necessário.

Não há uma maneira ideal para fazer isso. Crie a sessão da maneira que achar necessário, apontando para um banco de dados de teste. Geralmente isso é feito em um método de inicialização do próprio teste:

```
private Session session;

@Before
public void inicializa() {
    // pegando uma conexão com banco de testes
    session = new CriadorDeSessoes().bancoDeTestes();
    session.beginTransaction();
}
```

Outro problema é limpar esse banco de dados. Imagine por exemplo um teste que conte a quantidade de produtos cadastrados. Se já tivermos algum produto ca-

dastrado antes da execução do teste, provavelmente o teste falhará, pois ele não contava com esse dado a mais. Portanto, cada teste precisa ter o banco de dados limpo para que dados antigos não influenciem no resultado.

É comum fazermos uso de métodos de finalização para isso. Esses métodos geralmente dão um simples rollback na transação. Veja um exemplo:

```
@After
public void limpaTudo() {
    session.getTransaction().rollback();
    session.close();
}
```

Novamente, testes de integração podem se tornar complicados. Por exemplo, caso um teste exija dados pré-salvos no banco de dados, é responsabilidade do teste de montar todo o cenário e persisti-lo.

> **Rollback ou truncar tabelas?**
>
> Dar um rollback nas tabelas é geralmente a solução mais simples, pois basta uma linha de código.
>
> Mas já vi testes passarem e em produção a funcionalidade não funcionar, justamente porque o banco recusava as inserções no momento do "commit da transação". Portanto, uma solução mais completa seria comitar a transação, e em seguida, truncar todas as tabelas. Naturalmente é uma solução que dá mais trabalho.

Testar o método `ativos()` também não é difícil. Precisamos montar um cenário onde o banco de dados contenha produtos ativos e não ativos. O retorno do método deve conter apenas os não ativos. Repare ali que, após instanciarmos os objetos, os salvamos explicitamente no banco de dados:

```
@Test
public void deveFiltrarAtivos() {
    ProdutoDao dao = new ProdutoDao(session);

    Produto ativo = new Produto("Geladeira", 150.0);
    Produto inativo = new Produto("Geladeira", 150.0);
    inativo.inativa();
```

```
    // salvando ambos os produtos no banco
    session.save(ativo);
    session.save(inativo);

    List<Produto> produtos = dao.ativos();

    assertEquals(1, produtos.size());
    assertEquals(inativo, produtos.get(0));
}
```

Portanto, lembre-se de testar seus DAOs batendo em um banco de dados real. Não use mocks. Garanta realmente que suas consultas SQLs funcionam da maneira como você espera. Use também sua criatividade para escrever o teste; basta ter em mente que é um teste como qualquer outro. Você precisa montar um cenário (usando as várias práticas já discutidas aqui, como Test Data Builders, etc), invocar um método no seu DAO, e encontrar uma maneira de garantir que seu banco respondeu corretamente.

10.4 Devo usar TDD em testes de integração?

Repare que na seção acima, não usamos TDD. Tínhamos o DAO já escrito e depois o testamos. Em classes que lidam com infra estrutura, esse é geralmente o comportamento padrão. TDD faz muito sentido quando queremos testar algoritmos ou projetos de classe complexos. Classes como DAOs geralmente não apresentam uma lógica complicada, mas sim apenas código que integra com outros sistema. No caso do DAO, seus métodos são geralmente compostos por uma SQL e uma invocação de método.

Como também discutido no capítulo sobre acoplamento, ao praticar TDD, o desenvolvedor acaba por criar interfaces que representam a interação com sistemas externos. Essas interfaces tendem a ser bem claras e específicas. Ao terminar a implementação da classe principal, o desenvolvedor parte para as classes ao redor, como DAOs e etc. Nessa hora, como não há muitas decisões a ser tomada, não há grandes problemas em implementar a classe e só depois escrever o teste.

Óbvio que, caso você esteja criando alguma integração mais complexa, TDD muito provavelmente te ajudará a entender melhor o problema e te guiará (através do já conhecido *feedback*) a uma solução melhor. Analise caso a caso.

10.5 Testes em aplicações Web

Novamente, a ideia de se escrever testes antes pode ser aplicada a qualquer contexto. Se o programador sentir que isso irá ajudar, então deve fazê-lo. Em sistemas web, geralmente a grande dificuldade de se escrever um teste é justamente separar as "camadas de integração" da camada de domínio.

Independente do framework escolhido para ajudar no desenvolvimento (Struts, Spring MVC, VRaptor, JSF, etc), em ambos o desenvolvedor é obrigado a escrever uma camada que "conecta" o mundo web, cheio de requisições, respostas, HTTP e HTML, com o mundo de domínio, cheio de classes de negócio ricas. Chamamos essa camada geralmente de "controlador" (ou Controller). Controllers devem ser adaptadores. Não devem possuir regras de negócio, apenas regras de fluxo e exibição.

Veja abaixo um exemplo de um método de um controlador de uma aplicação escrita com VRaptor, responsável por decidir qual o próximo passo do usuário em um sistema de ensino online. Se o aluno já terminou o curso, o controlador redireciona para o certificado, se ele não terminou, redireciona para a próxima seção que ele deve fazer, e etc:

```
@Get("/course/{course.code}/continue")
public void continueCourse(Course course) {
    course = courses.findBy(course.getCode());
    if(enrollments.getEnrollmentFor(loggedUser, course).
                            hasAchievedCertification()) {

        result.redirectTo(UserController.class)
            .certificate(loggedUser, course.getCode());

    } else if (course.isPreSale()) {
        result.redirectTo(HomeController.class).dashboard();
    } else {
        Section section =
            enrollments.getEnrollmentFor(loggedUser, course)
            .continuing();

        result.redirectTo(SectionController.class)
            .show(section.getCourse().getCode(), section.getNumber(),
                 0);
    }
}
```

Veja que não há regras de negócio no código acima. As regras de negócio estão isoladas em classes de domínio (e, como mostrado ao longo do livro, são facilmente testadas) e acesso a infra estrutura, como banco de dados, também está isolado (e podem ser testados também, como discutido nesse capítulo). Ao isolar todo código de negócio da camada de controle, o desenvolvedor consegue testar a "parte que importa"

Agora, um ponto interessante a ser discutido é sobre a necessidade de se testar um controlador ou não. Muitos desenvolvedores gostam de testar controladores afim de garantir que eles sempre tomarão a decisão certa sobre a próxima página a ser exibida.

Mas veja o código acima. Para testá-lo, são necessários muitos mocks: *courses* é um DAO que busca cursos, *enrollments* é um DAO que busca por inscrições em cursos, *result* é responsável por redirecionar para outra JSP ou método. Um simples teste fará uso de diversos mocks. Já discutimos isso anteriormente: isso mostra o alto acoplamento dessa classe.

Controladores são um bom exemplo de classes cujo acoplamento alto é aceitável. De novo, eles são adaptadores, e portanto se conectam a dois mundos diferentes, cada um com sua classe. Já discutimos também sobre como testes que usam mocks são altamente acoplados com a implementação. Eles sabem como a implementação funciona, e portanto qualquer mudança na implementação impacta no teste. São testes frágeis.

Geralmente, eu opto por não escrever testes de unidade para controladores. Acho esses testes muito frágeis, mudam o tempo inteiro e não dão o *feedback* esperado. Bugs em controladores geralmente acontecem não pelo seu código em si, mas por problemas de integração com a camada web. Nomes de elementos no HTML que são submetidos pelo formulário e capturados pelo controlador são causadores de muitos bugs.

E sim, essa integração deve ser testada. Mas, em minha opinião, a melhor solução é escrever um teste de sistema. Ou seja, deve-se subir o browser, navegar na aplicação web, submeter formulários e validar a saída. Para testes como esses, ferramentas como Selenium (que sabem como abrir o browser e manipular os elementos) são de grande valia.

Fazer TDD em testes de sistema tem sido estudado pela comunidade acadêmica há algum tempo. Eles batizaram a técnica de ATDD (*Acceptance Test-Driven Development*). A ideia é escrever um teste de sistema que falha, em seguida fazer o ciclo de TDD para criar todas as classes necessárias para implementar a funcionalidade,

e por consequência, fazer o teste de sistema passar. Na prática, não vejo como fazer isso funcionar de maneira produtiva. Um teste de sistema é um teste caro e frágil de ser escrito. Gosto deles, mas escritos após a implementação.

10.6 Conclusão

Não há uma receita mágica para escrever testes de integração. Mas lembre-se que, caso você tenha uma classe cuja responsabilidade é justamente se integrar com outro sistema, você deve fazer um teste de integração.

Consiga uma réplica do sistema ou alguma maneira de consumi-lo para testes, seja ele um banco de dados, um serviço web, ou qualquer outra coisa.

O grande desafio, no fim, é conseguir montar o cenário e validar a saída nesse serviço externo. Talvez você escreva um "teste feio", com muitas linhas de código. Mas não se preocupe, seu sistema estará bem testado e você dormirá em paz.

Capítulo 11

Quando não usar TDD?

Ao longo deste livro, discutimos as diversas vantagens da escrita de testes de unidade e TDD, tanto na qualidade externa quanto na qualidade interna do sistema.

Mas será que devemos fazer uso da prática 100% do tempo? Neste capítulo discutimos quando usar e, mais importante, quando não usar TDD.

11.1 Quando não praticar TDD?

Repetindo uma frase que apareceu diversas vezes ao longo do livro. Em engenharia de software, não se deve usar as palavras "nunca" e "sempre". Nenhuma prática deve ser usada 100% do tempo ou descartada de uma vez.

Em momentos onde o desenvolvedor não sabe bem como resolver o problema, dado sua complexidade, ou não sabe bem como projetar uma classe, dado sua necessidade de ser flexível, TDD pode ser de grande valia devido ao seu constante *feedback* sobre a qualidade do código.

Mas, em momentos onde o desenvolvedor já sabe como resolver o problema, tanto do ponto de implementação, quanto do ponto de vista de design, não utilizar

TDD pode não ser um problema tão grande assim. Uma outra situação onde o uso de TDD pode ser desnecessário é durante a escrita de códigos que lidam com infra estrutura ou com camada de visualização. Durante a implementação de um DAO, por exemplo, não há grande necessidade de *feedback* em relação a qualidade do código (pois é geralmente uma SQL) e do projeto de classes (geralmente DAOs são classes cujas decisões de design são simples).

Novamente, a ideia não é usar TDD porque a prática está em alta no mercado, mas sim pelo benefício que ele agrega ao processo. Se não há a necessidade de *feedback* constante naquele momento, não há necessidade de se praticar TDD. O que talvez seja difícil é justamente julgar corretamente quais são esses momentos. A experiência do desenvolvedor, em desenvolvimento de software e em TDD, são cruciais.

Outro ponto importante a ser levantado é que estamos discutindo a não utilização de TDD em alguns momentos, mas não a falta de testes. Caso o desenvolvedor opte por não usar TDD, espera-se que ele, após a implementação, escreva testes para aquela funcionalidade, sejam eles de unidade ou integração. É preciso garantir a qualidade externa de maneira sustentável, e como discutido na introdução, essa maneira é automatizando o teste.

Deixe sua experiência te guiar também na escolha de usar ou não TDD. Lembre-se que no fim das contas, o que precisamos é de *feedback* sobre o que estamos fazendo.

Você não faz TDD 100% do tempo então?

Não. Dou preferência sim para TDD, mas nos casos discutidos acima, geralmente opto por não fazer. Mas, mesmo quando não faço, tento ao máximo aumentar a quantidade de *feedback* que recebo dos testes que escrevo após a implementação.

Ou seja, mesmo não praticando TDD, tento sempre escrever uma pequena quantidade de código e em seguida um teste. E repito.

11.2 100% de cobertura de código?

Cobertura de código é a métrica que nos diz a quantidade de código que está testada e a quantidade de código onde nenhum teste o exercita. Uma cobertura de código de 100% indica que todo código de produção tem ao menos um teste passando por ele.

Muitas pessoas discutem a necessidade de ter 100% de cobertura em um código. Mas na verdade, não há problemas em códigos que não tenham 100% de cobertura de testes de unidade. Uma equipe deve ter como meta buscar sempre a maior cobertura possível; mas essa é uma meta que provavelmente ela não irá alcançar. Alguns trechos de código simplesmente não valem a pena serem testados de maneira isolada, ou por serem simples demais.

Veja essa classe abaixo, extraída do projeto Restfulie.NET, por exemplo, chamada AspNetMvcUrlGenerator. Ela serve para gerar URLs para Actions em Controllers, utilizando as rotas pré-definidas. Repare que ela faz uso intenso das APIs do Asp.Net MVC (framework para desenvolvimento Web da Microsoft), utilizando inclusive alguns métodos estáticos (que já sabemos que são difíceis de serem testados) como no `HttpContext`.

```
public class AspNetMvcUrlGenerator : IUrlGenerator
    {
        public string For(string controller, string action,
                                            IDictionary values)
        {
            var httpContextWrapper =
                    new HttpContextWrapper(HttpContext.Current);

            var urlHelper =
                    new UrlHelper(new RequestContext(httpContextWrapper,
              RouteTable.Routes.GetRouteData(httpContextWrapper)));

            return FullApplicationPath(httpContextWrapper.Request) +
              urlHelper.Action(action, controller,
                               new RouteValueDictionary(values));
        }

        private string FullApplicationPath(HttpRequestBase request)
        {
            var url = request.Url.AbsoluteUri.Replace(
                request.Url.AbsolutePath, string.Empty)
                + request.ApplicationPath;

            return url.EndsWith("/") ?
                url.Substring(0, url.Length - 1) : url;
        }
    }
```

Como testar essa classe de maneira isolada? Só através de mágica para conseguir simular as mais diversas classes do próprio Asp.Net Mvc. Mas, mesmo que possível, a única vantagem de se testar essa classe seria para aumentar a métrica de cobertura. Esse trecho de código precisa de um teste de integração, e não de um teste de unidade.

Um outro exemplo de código que não vale a pena ser testado são getters e setters. Geralmente eles são gerados pelo Eclipse, e existem aos milhares pelo sistema. Escrever testes exclusivos para eles é pura perda de tempo.

Um desenvolvedor deve cobrir seu código de testes, mas pode usar testes de diferentes níveis para isso (testes de unidade, de integração, de sistema). Escrever testes de unidade inúteis, apenas para chegar nos 100% de cobertura, é desperdício.

Mas como fazer essa métrica ser útil?

Configure a métrica para ignorar getters e setters, e todas as outras classes que você já sabe que não escreverá testes. Desse jeito, a métrica pode passar a ser mais útil.

11.3 Devo testar códigos simples?

Muitos desenvolvedores possuem essa heurística: "se o código é simples, então não preciso testá-lo". A regra pode até fazer algum sentido, mas ela é incompleta.

Alguns trechos de código não merecem ser testados mesmo. Getters e setters, por exemplo, que geralmente são gerados automaticamente pela sua IDE favorita, não precisam de testes próprios. Alguns construtores (que muitas vezes também são gerados automaticamente) também não precisam ser testados. Métodos que repassam a invocação do seu método para um método de uma de suas dependências e por isso só possuem uma única linha talvez também não precisem ser testados.

Mas cuidado para não achar que todo método é simples. Muitos bugs caros são gerados por apenas uma linha de código. Lembre-se que no momento em que o desenvolvedor escreve o código, ele parece simples aos olhos dele. Mas é fato que esse código sofrerá alterações no futuro; e muito provavelmente por um outro desenvolvedor. Como garantir que as alterações feitas não fazem com que as regras de negócio antigas parem de funcionar? Testes são necessários.

Portanto, cuidado na hora de julgar se um código merece ou não ser testado. Na dúvida, teste. Lembre-se que todo código é considerado culpado até que se prove inocente.

11.4 Erros comuns durante a prática de TDD

Ao praticar TDD, desenvolvedores às vezes cometem alguns deslizes que podem prejudicar o *feedback* da prática [14].

Escrever o teste e não vê-lo falhar pode não ser uma boa ideia. Se o desenvolvedor escreve um teste achando que ele falharia, mas ele passa, algo está errado. Ou ele não entende muito bem o código de produção que existe, ou o teste não está bem implementado.

A ideia de também ver o teste passar rapidamente é justamente para que você "teste o teste". Teste automatizado é código; e esse código pode conter bugs. Ao ver o teste passar, você garantiu que o teste fica vermelho ou verde na hora certa.

Refatorar também é um passo importante do ciclo de TDD. Como discutido ao longo do livro, TDD faz com que você escreve código simples frequentemente. Mas o simples pode não ser a melhor solução para o problema. Para chegar nela, o desenvolvedor deve refatorar seu código constantemente. Em um de meus estudos, observei que a grande maioria dos desenvolvedores diz esquecer da etapa de refatoração e partir logo para o próximo teste de maneira frequente. Isso pode não ser uma boa ideia.

Não começar pelo cenário mais simples e ir evoluindo aos poucos também pode gerar código complexo desnecessário. Ao começar pelo cenário simples, o desenvolvedor garante que irá escrever a menor quantidade de código para fazer o teste passar. Evoluir aos poucos ajuda o desenvolvedor a aprender mais sobre o problema e ir evoluindo o código de maneira mais sensata.

Os passos de TDD fazem sentido. Mas, novamente, use seu bom senso e experiência na hora de decidir usá-los ou não.

11.5 Como convencer seu chefe sobre TDD?

Geralmente um programador que nunca praticou TDD tem essa dúvida: será que TDD realmente ajuda na qualidade do código? E na redução de defeitos? Ele aumenta ou diminui a produtividade, afinal? Mas como toda e qualquer prática em engenharia de software, é muito difícil avaliar e chegar a uma conclusão exata sobre seus ganhos e benefícios.

Ao longo do livro, citamos diversas vantagens da escrita de testes e TDD. Maior qualidade externa, interna (do ponto de vista de implementação e projeto de classes), produtividade (testes manuais no fim saem mais caros), entre outras.

Nos últimos anos, a comunidade acadêmica vem rodando diversos experimentos para tentar mostrar de maneira empírica que TDD realmente ajuda no processo de desenvolvimento de software. Alguns desses estudos são feitos por professores bastante conhecidos na comunidade, como a prof. Laurie Williams (North Carolina State University) e o prof. David Janzen (California Polytechnic State University).

Algumas dessas pesquisas investigam o fato de TDD reduzir o número de defeitos de um software; já outros investigam o fato de TDD produzir código de melhor qualidade. Alguns até pesquisam por indícios de aumento de produtividade. Abaixo listo alguns desses estudos que podem ser usados para convencer seu chefe.

Estudos na indústria

Janzen [21] mostrou que programadores usando TDD na indústria produziram código que passaram em aproximadamente 50% mais testes caixa-preta do que o código produzido por grupos de controle que não usavam TDD. Além do mais, o grupo que usava TDD gastou menos tempo debugando. Janzen também mostrou que a complexidade dos algoritmos era muito menor e a quantidade e cobertura dos testes era maior nos códigos escritos com TDD.

Um estudo feito pelo Maximillien e Williams [13] mostrou uma redução de 40-50% na quantidade de defeitos e um impacto mínimo na produtividade quando programadores usaram TDD.

Outro estudo feito por Lui e Chan [10] comparando dois grupos, um utilizando TDD e o outro escrevendo testes apenas após a implementação, mostrou uma redução significante no número defeitos. Além do mais, os defeitos que foram encontrados eram corrigidos mais rapidamente pelo grupo que utilizou TDD. O estudo feito por Damm, Lundberg e Olson [11] também mostra uma redução significante nos defeitos.

O estudo feito por George e Williams [12] mostrou que, apesar de TDD poder reduzir inicialmente a produtividade dos desenvolvedores mais inexperientes, o código produzido passou entre 18% a 50% mais em testes caixa-preta do que códigos produzidos por grupos que não utilizavam TDD. Esse código também apresentou uma cobertura entre 92% a 98%. Uma análise qualitativa mostrou que 87.5% dos programadores acreditam que TDD facilitou o entendimento dos requisitos e 95.8% acreditam que TDD reduziu o tempo gasto com debug. 78% também acreditam que TDD aumentou a produtividade da equipe. Entretanto, apenas 50% acreditam que TDD ajuda a diminuir o tempo de desenvolvimento. Sobre qualidade, 92% acreditam que TDD ajuda a manter um código de maior qualidade e 79% acreditam que

ele promove um design mais simples.

Nagappan [29] mostrou um estudo de caso na Microsoft e na IBM e os resultados indicaram que o número de defeitos de quatro produtos diminuir entre 40% a 90% em relação à projetos similares que não usaram TDD. Entretanto, o estudo mostrou também TDD aumentou o tempo inicial de desenvolvimento entre 15% a 35%.

Langr [22] mostrou que TDD aumenta a qualidade código, provê uma facilidade maior de manutenção e ajuda a produzir 33% mais testes comparados a abordagens tradicionais.

Estudos na academia

Um estudo feito por Erdogmus et all [20] com 24 estudos de graduação mostrou que TDD aumenta a produtividade. Entretanto nenhuma diferença de qualidade no código foi encontrada.

Outro estudo feito por Janzen[8] com três diferentes grupos de alunos (cada um deles usando uma abordagem diferente: TDD, test-last, sem testes), mostrou que o código produzido pelo time que fez TDD usou melhor conceitos de orientação a objetos e as responsabilidades foram separadas em 13 diferentes classes enquanto que os outros times produziram um código mais procedural. O time de TDD também produziu mais código e entregou mais funcionalidades. Os testes produzidos por esse time teve duas vezes mais asserções que os outros e cobriu 86% mais branches do que o time test-last. Além do mais, as classes testadas tinham valores de acoplamento 104% menor do que as classes não testadas e os métodos eram, na média, 43% menos complexos do que os não-testados.

O estudo de Müller e Hagner [15] mostrou que TDD não resulta em melhor qualidade ou produtividade. Entretanto, os estudantes perceberam um melhor reuso dos códigos produzidos com TDD. Steinberg [31] mostrou que código produzido com TDD é mais coeso e menos acoplado. Os estudantes também reportaram que os defeitos eram mais fáceis de serem corrigidos. O estudo do Edwards [17] com 59 estudantes mostrou que código produzido com TDD tem 45% menos defeitos e faz com que o programador se sinta mais a vontade com ele.

11.6 TDD em Sistemas Legados

A grande dificuldade de se praticar TDD (e testes de unidade) em sistemas legados é que geralmente eles apresentam uma grande quantidade de código procedural. Códigos procedurais geralmente contém muitas linhas de código, lidam com diferentes

responsabilidades, acessam banco de dados, modificam a interface do usuário, o que os tornam difíceis de serem testados.

Nesses casos, a sugestão é para que o desenvolvedor, antes de começar a refatorar o legado, escreva testes automatizados da maneira que conseguir. Se conseguir escrever um teste de unidade, excelente. Mas caso não consiga, então ele deve começar a subir o nível até conseguir escrever um teste. Em aplicações legadas web, por exemplo, o desenvolvedor pode começar com testes de sistema, para garantir o comportamento do sistema.

Com esses testes escritos, o desenvolvedor conseguirá refatorar o código legado problemático que está por baixo, e ao mesmo tempo garantir a qualidade do que está fazendo. Nessa refatoração, o desenvolvedor deve criar classes menores e mais coesas e já escrever testes de unidade para elas.

Durante essa reescrita, pode ser que o desenvolvedor sinta a necessidade de escrever testes "feios", como testes para testar *stored procedures* ou coisas do tipo. Nesses casos, escreva o teste, implemente a nova solução, elimine a *stored procedure* bem como o seu teste. Em sistemas legados, não tenha medo de escrever código de teste que será jogado fora depois. Se fizermos uma analogia com a engenharia nesse momento, imagine que esses códigos sejam como os andaimes utilizados durante a obra. Eles não fazem parte do produto final, estão lá apenas para ajudar no desenvolvimento.

Por mim, essa é a sugestão. Escreva testes para o sistema legado da maneira que conseguir, refatore o código já criando uma bateria de testes amigável. Ao final, caso faça sentido, jogue fora o código legado bem como testes que não são mais úteis.

A dificuldade em se testar sistemas legados não se aplica para novas funcionalidades. Nelas, ele tem a chance de escrever um código melhor e mais fácil de ser testado. Alguns padrões de projeto, como o Adapter, ajudam o desenvolvedor a "conectar" código bem escrito com o legado que deve ser mantido.

11.7 Conclusão

TDD é uma excelente ferramenta, deve constar no cinto de utilidades de todo desenvolvedor, mas obviamente deve ser usada nos momentos certos. Está em um momento complicado do desenvolvimento daquela classe? Use TDD. Está em um momento onde não há muitas dúvidas sobre o que fazer? Não se sinta mal por escrever testes após a implementação.

Lembre-se, o objetivo final é escrever código de qualidade.

Capítulo 12

E agora?

Agora que TDD já foi discutido sob todos os pontos de vista, qual o próximo passo? Neste capítulo, finalizo a discussão apontando para outras excelentes referências sobre o assunto, e quais os próximos passos para o praticante.

12.1 Como aprender mais agora?

A referência mais famosa sobre TDD é o livro do Kent Beck, intitulado "Test-Driven Development: By Example". Ele é um livro bem introdutório sobre TDD. Ele discute aos poucos como escrever os testes antes, como ser simples, o que são baby steps, e etc. Ao longo do livro ele cria uma classe `Dinheiro`, que realiza operações financeiras em unidades monetárias distintas. Talvez seja uma ótima referência para quem nunca leu nada sobre o assunto, e quer começar.

Um excelente livro sobre o assunto é o famoso "Growing Object Oriented Software, Guided by Tests", escrito por Steve Freeman e Nat Pryce. Neste livro, os autores mostram como utilizar TDD pode ajudar no design de suas classes. Ambos autores

são famosos na comunidade por suas diversas contribuições para a comunidade ágil e código aberto. É um livro que vale a pena ser lido.

Falando mais sobre design de classes, um bom livro sobre o assunto é o "Agile Software Development, Principles, Patterns, and Practices" do Robert Martin. Nele são discutidos os vários princípios de orientação a objetos de maneira profunda. Este livro contém um apêndice sobre o assunto, mas com certeza o livro do Uncle Bob é uma excelente referência para quem deseja se aprofundar em design orientado a objetos.

Por fim, aponto também para o blog da Caelum (http://blog.caelum.com.br) e meu blog pessoal (http://www.aniche.com.br) , onde tenho escrito bastante conteúdo sobre o assunto. Os posts, aliás, foram uma grande motivação para a escrita desse livro.

Caso você prefira uma abordagem com cursos presenciais, certamente os cursos da Caelum, os quais ajudei a criar, são uma excelente fonte de aprendizado:

http://www.caelum.com.br/cursos/agile/

12.2 Dificuldade no aprendizado

Nos últimos anos, tenho dado diversas aulas e workshops sobre o assunto, e hoje posso afirmar que a maior dificuldade na hora de se praticar TDD são:

- **Falta de conhecimento em orientação a objetos**. Quando o desenvolvedor não conhece orientação a objetos a fundo, ele sente dificuldades no momento da escrita do teste. Entender bem os conceitos e princípios do paradigma o ajudará a aprender TDD mais facilmente.

- **Pensar em cenários para os testes**. Pensar em cenários não é algo tão natural. Lembre-se que um teste automatizado é muito parecido com um teste manual. Imagine quais os cenários você testaria em sua aplicação como um todo, e vá descendo de nível até imaginar como aquela classe deve responder de acordo com as diferentes entradas.

Como toda nova prática introduzida em um ambiente de trabalho, é normal que a produtividade caia no começo. Para que TDD torne-se natural para a equipe, é necessário que ela pratique constantemente. Existem diversos exercícios e vídeos explicativos para que o desenvolvedor consiga praticar e dominar melhor a prática.

Junto com a Caelum, escrevi um curso online sobre o assunto. Ele pode ser encontrado no próprio site da Caelum (http://www.caelum.com.br/online) . Lá, produzi um conjunto de vídeos bem didáticos sobre a utilização de TDD em um sistema de leilões. Talvez essa seja uma ótima maneira para você treinar sua equipe também.

12.3 Como interagir com outros praticantes?

Como falado no começo do livro, criei um grupo de discussão no Google Groups. Você pode se cadastrar em https://groups.google.com/forum/#!forum/tdd-no-mundo-real.

Também abuse do GUJ, uma comunidade de perguntas e respostas para tirar dúvidas técnicas: http://www.guj.com.br/

12.4 Conclusão Final

Testar é vital para a qualidade de qualquer sistema. Lembre-se que qualidade é algo que não podemos abrir mão, dado a importância que sistemas de computador tem hoje perante a sociedade. Ao longo deste livro, tentei mostrar como pode ser fácil e divertido a escrita de testes automatizados. Espero ter mostrado também que a busca por testabilidade acaba guiando o programador a um design de classes melhor e mais sustentável.

Agora depende de você. É hora de escrever testes!

Capítulo 13

Apêndice: Princípios SOLID

Ao longo do curso, discutimos diversos princípios de design, de forma às vezes informal. Este apêndice serve para então consolidar os princípios apresentados ao longo do livro. Para maiores informações sobre eles, é sugerido que o leitor busque os trabalhos do Robert Martin [26].

13.1 Sintomas de Projetos de Classes em Degradação

Diz-se que um projeto de classes está *degradando* quando o mesmo começa a ficar difícil de evoluir, o reúso de código se torna mais complicado do que repetir o trecho de código, ou o custo de se fazer qualquer alteração no projeto de classes se torna alto.

Robert Martin enumerou alguns sintomas de projeto de classes em degradação, chamados também de *maus cheiros* de projeto de classes. Esses sintomas são parecidos com os maus cheiros de código *(code smells)*, mas em um nível mais alto: eles estão presentes na estrutura geral do software ao invés de estarem localizados em apenas um pequeno trecho de código.

Esses sintomas podem ser medidos de forma subjetiva e algumas vezes de forma até objetiva. Geralmente, esses sintomas são causados por violações de um ou mais princípios de projeto de classes. Muitos desses problemas são relacionados à gerência de dependências. Quando essa atividade não é feita corretamente, o código gerado torna-se difícil de manter e reusar. Entretanto, quando bem feita, o software tende a ser flexível, robusto e suas partes reusáveis.

Rigidez

Rigidez é a tendência do software em se tornar difícil de mudar, mesmo de maneiras simples. Toda mudança causa uma cascata de mudanças subsequentes em módulos dependentes. Quanto mais módulos precisam ser modificados, maior é a rigidez do projeto de classes.

Quando um projeto de classes está muito rígido, não se sabe com segurança quando uma mudança terá fim. Mudanças simples passam a demorar muito tempo até serem aplicadas no código e frequentemente acabam superando em várias vezes a estimativa de esforço inicial. Frases como *"isto foi muito mais complicado do que eu imaginei"* tornam-se populares. Neste momento, gerentes de desenvolvimento começam a ficar receosos em permitir que os desenvolvedores consertem problemas não críticos.

Fragilidade

Fragilidade é a tendência do software em quebrar em muitos lugares diferentes toda vez que uma única mudança acontece. Frequentemente, os novos problemas ocorrem em áreas não relacionadas conceitualmente com a área que foi mudada, tornando o processo de manutenção demasiadamente custoso, complexo e tedioso.

Consertar os novos problemas usualmente passa a resultar em outros novos problemas e assim por diante. Infelizmente, módulos frágeis são comuns em sistemas de software. São estes os módulos que sempre aparecem na lista de defeitos a serem corrigidos. Além disto, desenvolvedores começam a ficar receosos de alterar certos trechos de código, pois sabem que estes estão tão frágeis que qualquer mudança simples fatalmente acarretará na introdução de problemas inesperados e de naturezas diversas.

Imobilidade

Imobilidade é a impossibilidade de se reusar software de outros projetos ou de

partes do mesmo projeto. Neste cenário, o módulo que se deseja reutilizar frequentemente tem uma bagagem muito grande de dependências e não possui código claro. Depois de muita investigação, os arquitetos descobrem que o trabalho e o risco de separar as partes desejáveis das indesejáveis são tão grandes, que o módulo acaba sendo reescrito ao invés de reutilizado.

Viscosidade

Quando uma mudança deve ser realizada, usualmente há várias opções para realizar tal mudança. Quando as opções que preservam o projeto de classes são mais difíceis de serem implementadas do que aquelas que não o preservam, há alta viscosidade de projeto de classes. Neste cenário, é fácil fazer a "coisa errada" e é difícil fazer a "coisa certa", ou seja, é difícil preservar e aprimorar o projeto de classes.

Complexidade Desnecessária

Detecta-se complexidade desnecessária no projeto de classes quando ele contém muitos elementos inúteis ou não utilizados (*dead code*). Geralmente ocorre quando há muito projeto inicial (*up-front design*) e não se segue uma abordagem de desenvolvimento iterativa e incremental, de modo que os projetistas tentam prever uma série de futuros requisitos para o sistema e concebem um projeto de classes demasiadamente flexível ou desnecessariamente sofisticado.

Frequentemente apenas algumas previsões acabam se concretizando ao longo do tempo e, neste meio período, o projeto de classes carrega o peso de elementos e construções não utilizados. O software então se torna complexo e difícil de ser entendido. Projetos com complexidade muito alta comumente afetam a produtividade, porque quando os desenvolvedores herdam tal projeto, eles gastam muito tempo aprendendo as nuances do projeto de classes antes que possam efetivamente estendê-lo ou mantê-lo confortavelmente.

Repetição Desnecessária

Quando há repetição de trechos de código, é sinal de que uma abstração apropriada não foi capturada durante o processo de projeto de classes (ou inclusive na análise). Esse problema é frequente e é comum encontrar softwares que contenham dezenas e até centenas de elementos com códigos repetidos.

Descobrir a melhor abstração para eliminar a repetição de código geralmente não está na lista de itens de alta prioridade dos desenvolvedores, de maneira que a

resolução do problema acaba sendo eternamente postergada. Também, o sistema se torna cada vez mais difícil de entender e principalmente de manter, pois os problemas encontrados em uma unidade de repetição devem ser corrigidos potencialmente em toda repetição, com o agravante de que uma repetição pode ter forma ligeiramente diferente de outra.

Opacidade

Opacidade é a tendência de um módulo ser difícil de ser entendido. Códigos podem ser escritos de maneira clara e expressiva ou de maneira "opaca" e complicada. A tendência de um código é se tornar mais e mais opaco à medida que o tempo passa e, para que isso seja evitado, é necessário um esforço constante em manter esse código claro e expressivo.

Uma maneira para prevenir isso é fazer com que os desenvolvedores se ponham no papel de leitores do código e refatorem esse código de maneira que qualquer outro leitor poderia entender. Além disso, revisões de código feita por outros desenvolvedores é também uma possível solução para manter o código menos opaco.

13.2 Princípios de Projeto de Classes

Todos os problemas citados na seção anterior podem ser evitados pelo uso puro e simples de orientação a objetos. A máxima da programação orientada a objetos diz que classes devem possuir um baixo acoplamento e uma alta coesão.

Alcançando esse objetivo, mudanças no código seriam executadas mais facilmente; alterações seriam feitas em pontos únicos e a propagação de mudanças seria bem menor. Com as abstrações bem estabelecidas, novas funcionalidades seriam implementadas através de novo código, sem a necessidade de alterações no código já existente. Necessidades de evolução do projeto de classes seriam feitas com pouco esforço, já que módulos dependeriam apenas de abstrações.

Mas, alcançar tal objetivo não é tarefa fácil. Criar classes pouco acopladas e altamente coesas demanda um grande esforço por parte do desenvolvedor e requer grande conhecimento e experiência no paradigma da orientação a objetos.

Os princípios comentados nesta seção são muito discutidos por Robert Martin em vários de seus livros e artigos publicados. Esses princípios são produto de décadas de experiência em engenharia de software. Segundo ele, esses princípios não são produto de uma única pessoa, mas sim a integração de pensamentos e trabalhos de um grande número de desenvolvedores de software e pesquisadores, e visam

combater todos os sintomas de degradação discutidos acima.

Conhecidos pelo acrônimo **SOLID** (sólido, em português), são eles:

- Princípio da Responsabilidade Única (Single-Responsibility Principle (SRP))
- Princípio do Aberto-Fechado (Open-Closed Principle (OCP))
- Princípio de Substituição de Liskov (Liskov Substitution Principle (LSP))
- Princípio da Inversão de Dependência (Dependency Inversion Principle (DIP))
- Princípio da Segregação de Interfaces (Interface Segregation Principle (ISP))

Princípio da Responsabilidade Única

O termo coesão define a relação entre os elementos de um mesmo módulo. Isso significa que os todos elementos de uma classe que tem apenas uma responsabilidade tendem a se relacionar. Diz-se que uma classe como essa é uma classe que possui alta coesão (ou que é coesa). Já em uma classe com muitas responsabilidades diferentes, os elementos tendem a se relacionar apenas em "grupos", ou seja, com os elementos que tratam de uma das responsabilidades da classe. A esse tipo de classe, diz-se que ela possui uma baixa coesão (ou que não é coesa). Robert Martin altera esse conceito de coesão e a relaciona com as forças que causam um módulo ou uma classe a mudar. No caso, o Princípio de Responsabilidade Única diz que uma classe deve ter apenas uma única razão para mudar.

Esse princípio é importante no momento em que há uma alteração em alguma funcionalidade do software. Quando isso ocorre, o programador precisa procurar pelas classes que possuem a responsabilidade a ser modificada. Supondo uma classe que possua mais de uma razão para mudar, isso significa que ela é acessada por duas partes do software que fazem coisas diferentes. Fazer uma alteração em uma das responsabilidades dessa classe pode, de maneira não intencional, quebrar a outra parte de maneira inesperada. Isso torna o projeto de classes frágil.

Princípio do Aberto-Fechado

O Princípio do Aberto-Fechado, cunhado por Bertrand Meyer, diz que as entidades do software (como classes, módulos, funções, etc) devem ser abertas para extensão, mas fechadas para alteração. Se uma simples alteração resulta em uma cascata de alterações em módulos dependentes, isso cheira à rigidez. O princípio pede

então para que o programador sempre refatore as classes de modo que mudanças desse tipo não causem mais modificações.

Quando esse princípio é aplicado de maneira correta, novas alterações fazem com que o programador adicione novo código, e não modifique o anterior. Isso é alcançado através da criação de abstrações para o problema. Linguagens orientadas a objetos possuem mecanismos para criá-las (conhecido com interfaces em linguagens como Java ou C#). Através dessas abstrações, o programador consegue descrever a maneira em que uma determinada classe deve se portar, mas sem se preocupar em como essa classe faz isso.

Princípio de Substituição de Liskov

Esse princípio, que discute sobre tipos e subtipos, criado por Barbara Liskov em 1988, é importante já que herança é uma das maneiras para se suportar abstrações e polimorfismo em linguagens orientadas a objetos e, como visto acima, o Princípio do Aberto-Fechado se baseia fortemente na utilização desses recursos.

O problema é que utilizar herança não é tarefa fácil, pois o acoplamento criado entre classe filha e classe pai é grande. Fazer as classes filhas respeitarem o contrato do pai, e ainda permitir que mudanças na classe pai não influenciem nas classes filhas requer trabalho.

O princípio de Liskov diz que, se um tipo S é subclasse de um tipo T, então objetos do tipo T podem ser substituídos por objetos do tipo S, sem alterar nenhuma das propriedades desejadas daquele programa.

Um clássico exemplo sobre Princípio de Substituição de Liskov é o exemplo dos Quadrados e Retângulos. Imagine uma classe Retângulo. Um retângulo possui dois lados de tamanhos diferentes. Imagine agora uma classe Quadrado (figura geométrica que possui todos os lados com o mesmo tamanho) que herde de Retângulo. A única alteração é fazer com que os dois lados tenham o mesmo tamanho. Apesar de parecer lógico, afinal um Quadrado é um Retângulo com apenas uma condição diferente, a classe Quadrado quebra o Princípio de Liskov: a pré-condição dela é mais forte do que a do quadrado, afinal os dois lados devem ter o mesmo tamanho.

Quebras do princípio de Liskov geralmente levam o programador a quebrar o princípio do OCP também. Ele percebe que, para determinados subtipos, ele precisa fazer um tratamento especial, e acaba escrevendo condições nas classes clientes que fazem uso disso.

Princípio da Inversão de Dependências

Classes de baixo nível, que fazem uso de infraestrutura ou de outros detalhes de implementação podem facilmente sofrer modificações. E, se classes de mais alto nível dependerem dessas classes, essas modificações podem se propagar, tornando o código frágil.

O Princípio de Inversão de Dependências se baseia em duas afirmações:

- Módulos de alto nível não devem depender de módulos de baixo nível. Ambos devem depender de abstrações
- Abstrações não devem depender de detalhes. Detalhes devem depender de abstrações

Em resumo, as classes devem, na medida do possível, acoplar-se sempre com módulos mais estáveis do que ela própria, já que, como as mudanças em módulos estáveis são menos prováveis, raramente essa classe precisará ser alterada por mudanças em suas dependências.

Princípio da Segregação de Interfaces

Acoplar-se com uma interface de baixa granularidade (ou gordas, do termo em inglês *fat interfaces*) pode ser perigoso, já que qualquer alteração que um outro cliente forçar nessa interface poderá ser propagada para essa classe.

O princípio da segregação de interfaces diz que classes cliente não devem ser forçados a depender de métodos que eles não usam. Quando uma interface não é coesa, ela contém métodos que são usados por um grupo de clientes, e outros métodos que são usados por outro grupo de clientes. Apesar de uma classe poder implementar mais de uma interface, o princípio diz que o cliente da classe deve apenas depender de interfaces coesas.

13.3 Conclusão

Todos os princípios discutidos na seção acima tentam diminuir os possíveis problemas de projeto de classes que possam eventualmente aparecer. Discutir o que é um bom projeto de classes é algo difícil; mas é possível enumerar algumas das características desejáveis: isolar elementos reusáveis de elementos não reusáveis, diminuir a propagação de alterações em caso de uma nova funcionalidade.

Um bom programador OO deve conhecer e aplicar esses princípios ao longo de sua base de código.

Referências Bibliográficas

[1] Kent Beck. Junit framework. http://www.junit.org.

[2] Kent Beck. Tdd 10 years later. http://junit.sourceforge.net/doc/testinfected/testing.htm.

[3] c2. Ten years of test driven development. http://c2.com/cgi/wiki?TenYearsOfTestDrivenDevelopment.

[4] c2. Test data builder. http://c2.com/cgi/wiki?TestDataBuilder.

[5] Philip Calçado. Evitando vos e bos. http://fragmental.com.br/wiki/index.php/Evitando_VOs_e_BOs.html.

[6] Paulo Caroli. Testing private methods, tdd, and test-driven refactoring. http://agiletips.blogspot.com/2008/11/testing-private-methods-tdd-and-test.html.

[7] Coding Dojo. Tdd 10 years later. http://www.codingdojo.org.

[8] D. Janzen e H. Saiedian. On the influence of test-driven development on software design. 2006.

[9] Caelum Ensino e Inovação. Formação online de testes automatizados. http://www.caelum.com.br/curso/online/testes-automatizados/.

[10] K. M. Lui e K. C. C. Chan. Test-driven development and software process improvement in china. 2004.

[11] L.-O. Damn e L. Lundberg. Introducing test automation and test-driven development: An experience report. 2005.

[12] B. George e L. Williams. An initial investigation of test- driven development in industry. 2003.

[13] E. M. Maximilien e L. Williams. Assessing test-driven development at ibm. 2003.

[14] Mauricio Aniche e Marco Aurélio Gerosa. Most common mistakes in test-driven development practice: Results from an online survey with developers. 2010.

[15] M. M. Müller e O. Hagner. Experiment about test-first programming. 2002.

[16] Michael Feathers e Steve Freeman. Tdd 10 years later. http://www.infoq.com/presentations/tdd-ten-years-later.

[17] S. H. Edwards. Using test-driven development in a classroom: Providing students with automatic, concrete feedback on performance. 2003.

[18] Michael Feathers. The deep synergy between testability and good design. http://michaelfeathers.typepad.com/michael_feathers_blog/2007/09/the-deep-synerg.html, 2008.

[19] et al Freeman, Bates. *Head First Design Patterns*. O'Reilly Media, 2004.

[20] et al. H. Erdogmus, M. Morisio. On the effectiveness of the test-first approach to programming. 2005.

[21] D. Janzen. Software architecture improvement through test-driven development. 2005.

[22] J. Langr. Evolution of test and code via test-first design. http://www.objectmentor.com/resources/articles/tfd.pdf, 2010.

[23] Delamaro Maldonado, Jino. *Introdução ao Teste de Software*. Editora Campus, 2007.

[24] Robert Martin. Stability. http://www.objectmentor.com/resources/articles/stability.pdf.

[25] Robert Martin. The transformation priority premise. http://cleancoder.posterous.com/the-transformation-priority-premise.

[26] Robert Martin. *Agile Practices, Patterns, and Practices in C*. Prentice Hall, 2006.

[27] Marco Aurélio Gerosa Mauricio Aniche, Thiago Ferreira. What concerns beginner test-driven development practitioners: A qualitative analysis of opinions in an agile conference. 2011.

[28] Mockito. Mockito. http://mockito.org.

[29] e T. Bhat N. Nagappan. Evaluating the efficacy of test-driven development: industrial case studies. 2006.

[30] Joel on Software. Things you should never do, part i. http://www.joelonsoftware.com/articles/fog0000000069.html.

[31] D. H. Steinberg. The effect of unit tests on entry points, coupling and cohesion in an introductory java programming course. 2001.

[32] Wikipedia. Cyclomatic complexity. http://en.wikipedia.org/wiki/Cyclomatic_complexity.

[33] Wikipedia. Numeração romana. http://pt.wikipedia.org/wiki/Numera%C3%A7%C3%A3o_romana.

[34] Computer World. Study: Buggy software costs users, vendors nearly 60b annually. http://www.computerworld.com/s/article/72245/Study_Buggy_software_costs_users_vendors_nearly_60B_annually.